中国开源软件推进联盟
英特尔公司 | ARM公司
清华大学 | 大连理工大学 | 兰州大学 | 北京工业大学
推荐用书





Analysis and Practice of μC/OS-III
Real Time Operating System

μC/OS-III
内核分析与应用开发

吴国伟 林驰 任健康 李照鑫◎编著
Wu Guowei Lin Chi Ren Jiankang Li Zhaoxin

清華大学出版社

清华开发者书库

μC/OS-Ⅲ内核分析与应用开发

Analysis and Practice of μC/OS-Ⅲ Real Time Operating System

吴国伟　林　驰　任健康　李照鑫　编著

Wu Guowei　Lin Chi　Ren Jiankang　Li Zhaoxin

清华大学出版社

北　京

内容简介

μC/OS-Ⅲ是一个基于优先级的可固化实时嵌入式操作系统内核，在各类嵌入式系统中有广泛的应用。本书对μC/OS-Ⅲ内核结构和各种机制进行了详细分析，并设置了应用场景，给出了基于μC/OS-Ⅲ的开发应用实例。全书共分10章，第1章介绍了μC/OS-Ⅲ的架构、组成及内核源码的关键数据结构和相互关系；第2章到第9章分别分析μC/OS-Ⅲ的任务管理机制、内核调度机制、任务间同步机制、中断管理、定时器管理、时钟管理、内存管理和文件系统，并给出每种机制的应用实例；第10章介绍了μC/OS-Ⅲ的移植方法。在对μC/OS-Ⅲ的每一部分机制的源码分析过程中，先介绍工作机制，然后提炼关键数据结构和相互关系，再结合关键数据结构和算法分析源码，最后给出应用实例，让读者明白原理及实际应用，达到理论和实战技能同步提升的效果。为方便教学和自学，所有章节配有思考题与习题，以方便慕课、微课、微视频、翻转课堂等现代教学资源的制作。

本书可作为软件工程、电子信息科学与技术、计算机科学与技术、电子信息工程、电气工程及自动化、测控技术与仪器等专业的教材和有关工程技术人员的参考用书。

图书在版编目(CIP)数据

μC/OS-Ⅲ内核分析与应用开发/吴国伟等编著. —北京：清华大学出版社，2017
(清华开发者书库)
ISBN 978-7-302-48806-4

Ⅰ. ①μ…　Ⅱ. ①吴…　Ⅲ. ①实时操作系统　Ⅳ. ①TP316.2

中国版本图书馆CIP数据核字(2017)第274563号

责任编辑：盛东亮
封面设计：李召霞
责任校对：李建庄
责任印制：董　瑾

出版发行：清华大学出版社
网　址：http://www.tup.com.cn，http://www.wqbook.com
地　址：北京清华大学学研大厦A座　**邮　编**：100084
社 总 机：010-62770175　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：北京富博印刷有限公司
装 订 者：北京市密云县京文制本装订厂
经　销：全国新华书店
开　本：186mm×240mm　**印　张**：12.5　**字　数**：281千字
版　次：2018年11月第1版　**印　次**：2018年11月第1次印刷
定　价：59.00元

产品编号：064815-01

前言

PREFACE

μC/OS-Ⅲ实时操作系统(Micro C/OS Three)是一个可升级、可固化、基于优先级的实时内核。它是源码公开的商用性实时操作系统内核,由 μC/OS-Ⅱ发展而来。μC/OS-Ⅲ是一个第 3 代系统内核,它对任务的个数无限制,支持现代的实时内核所期待的大部分功能,例如资源管理、同步、任务间的通信等。同时,μC/OS-Ⅲ提供的特色功能在其他的实时内核中是找不到的,例如完备的运行时间测量功能,直接发送信号或者消息到任务,任务可以同时等待多个内核对象等。

第一代 μC/OS 系列产生于 1992 年。经过了多年的使用和上千人的反馈,已经产生了很多进化版本。μC/OS-Ⅲ是这些反馈和经验的总结。在 μC/OS-Ⅱ中很少使用的功能已经被删除或者被更新,增加了更高效的功能和服务。其中最有用的功能是时间片轮转法(round robin),这是 μC/OS-Ⅱ中不支持的。μC/OS-Ⅲ提供了新的功能以更好地适应新出现的处理器。特别地,μC/OS-Ⅲ被设计用于 32 位处理器,并且它也能在 16 位或 8 位处理器中很好地工作。

μC/OS-Ⅲ最主要的目标是提供一流的实时内核以适应快速更新的嵌入式产品。使用像 μC/OS-Ⅲ这样具有雄厚基础和稳定框架的商业实时内核,能够帮助设计师们处理日益复杂的嵌入式设计。μC/OS-Ⅲ实时操作系统具有高度的可移植性,能够移植到 ARM、Intel 等众多 CPU 上运行。因此,了解和学习 μC/OS-Ⅲ的运行原理是非常重要的。

本书面向的读者既包括需要使用 μC/OS-Ⅲ作底层操作系统,在其上进行应用开发的嵌入式应用开发人员,也包括想要了解 μC/OS-Ⅲ运行机制的学生或者开发人员。本书按照 μC/OS-Ⅲ的功能模块进行划分,对 μC/OS-Ⅲ的源码进行了详细介绍,同时在每一章的末尾,给出了具体的应用案例,读者可以选择先查看应用案例,了解 μC/OS-Ⅲ基本的应用程序调用接口(API),再在源码中查看 API 的相应实现。也可以先了解应用程序调用接口的实现机制,再去应用案例中借助 API 进行应用编程。

在本书撰写过程中,林驰和任健康编写第 1、3、5、6、8 章,李照鑫编写第 2、4、7 章,同时负责实验的设计和实现,吴国伟编写第 9、10 章。编写过程中研究生王志远、秦钰根和本科生游文华等做了大量的书稿校对和画图等工作。

希望各位读者在阅读本书时,能够思考 μC/OS-Ⅲ实时操作系统的机制与思想,这对于

自身提高有非常大的帮助。同时也希望各位读者，不要局限于书中内容，可以到 μC/OS-Ⅲ的官方网站，下载 μC/OS-Ⅲ源码的官方文档，同步学习。本书参考了很多书籍和网络资源，限于篇幅参考文献未一一列出，在此向作者表示感谢。如果发现书中有任何问题，请及时与我们联系，进行批评指正，我们也会及时地进行改正。

吴国伟

2018 年 7 月

目录

CONTENTS

第 1 章



μC/OS-Ⅲ操作系统概述

1.1 μC/OS-Ⅲ概览

μC/OS-Ⅲ操作系统是一个可裁剪、可升级、可固化、基于优先级的实时内核。它对任务的个数没有限制。μC/OS-Ⅲ是第 3 代系统内核，支持现代的实时内核所期待的大部分功能，例如资源管理、同步、任务间的通信等。并且，μC/OS-Ⅲ提供的特色功能在其他的实时内核中是找不到的，比如说完备的运行时间测量功能，直接地发送信号或者消息到任务，任务可以同时等待多个内核对象等。

μC/OS-Ⅲ操作系统的源代码开放，用户可以通过阅读其源代码来了解系统内部的实现细节，对系统有更好的了解。

μC/OS-Ⅲ内核由 C 语言编写，其源代码由 2 个.h 头文件和 17 个.c 文件构成，如图 1.1 所示。接下来将就每个源文件的功能进行介绍。

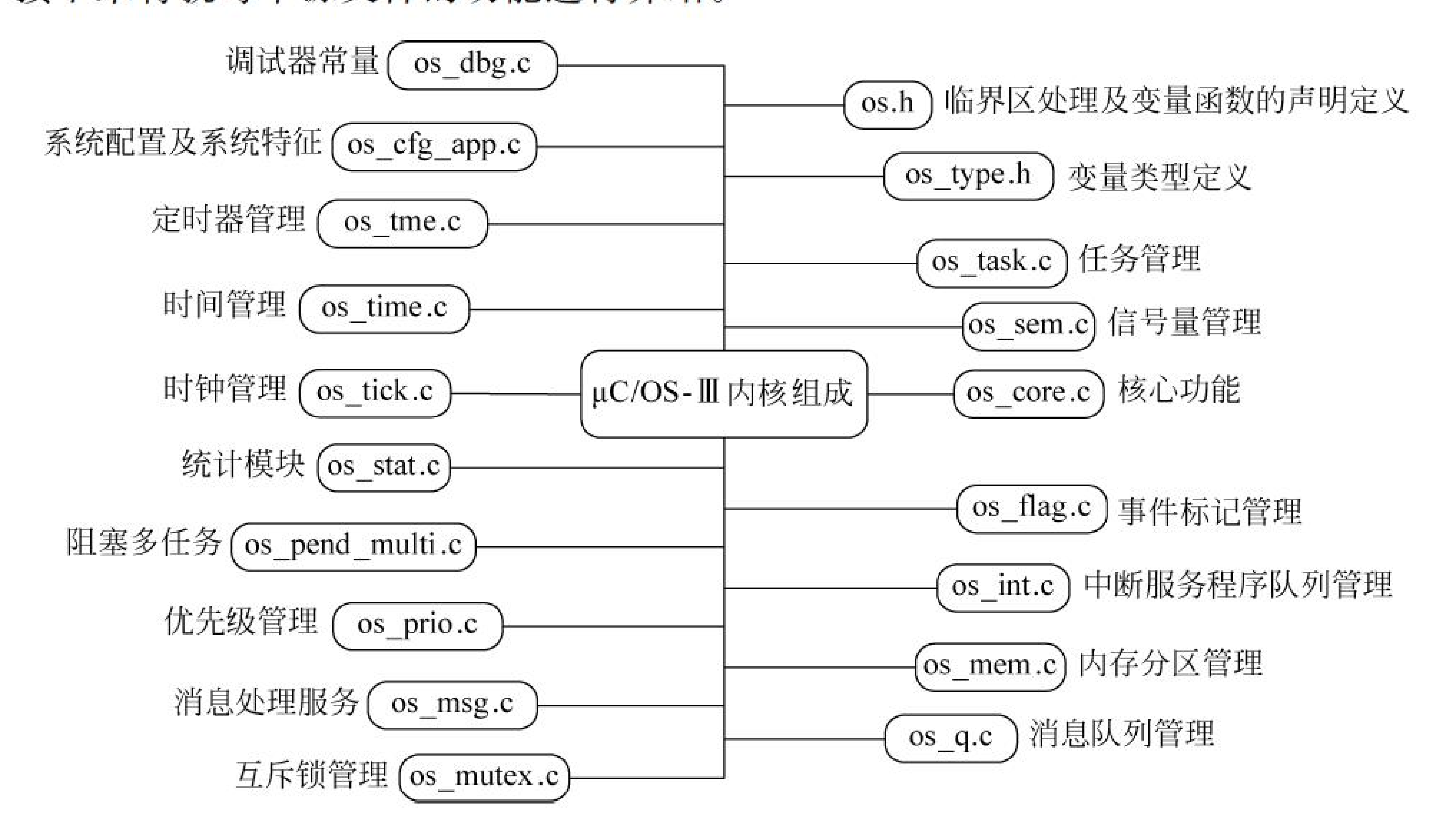


图 1.1　组成内核的各文件

1.1.1 os.h 和 os_type.h 功能

os.h 中包含了 μC/OS-Ⅲ中主要的头文件，定义了全局常量，还定义了提供给其他各个模块的主要的数据结构，而且定义了临界区进入和退出所采取的操作。临界区是内部不可打断的代码段，如果可能被中断打断，则需关闭中断，如果可能被其他任务打断，则需锁定调度器。

os_type.h 定义了各个自定义的数据类型，以实现在特定的模块中使用特定名称来表现数据类型，以便在移植时可更好地适应 CPU 架构。

os.h 和 os_type.h 的功能如图 1.2 所示。

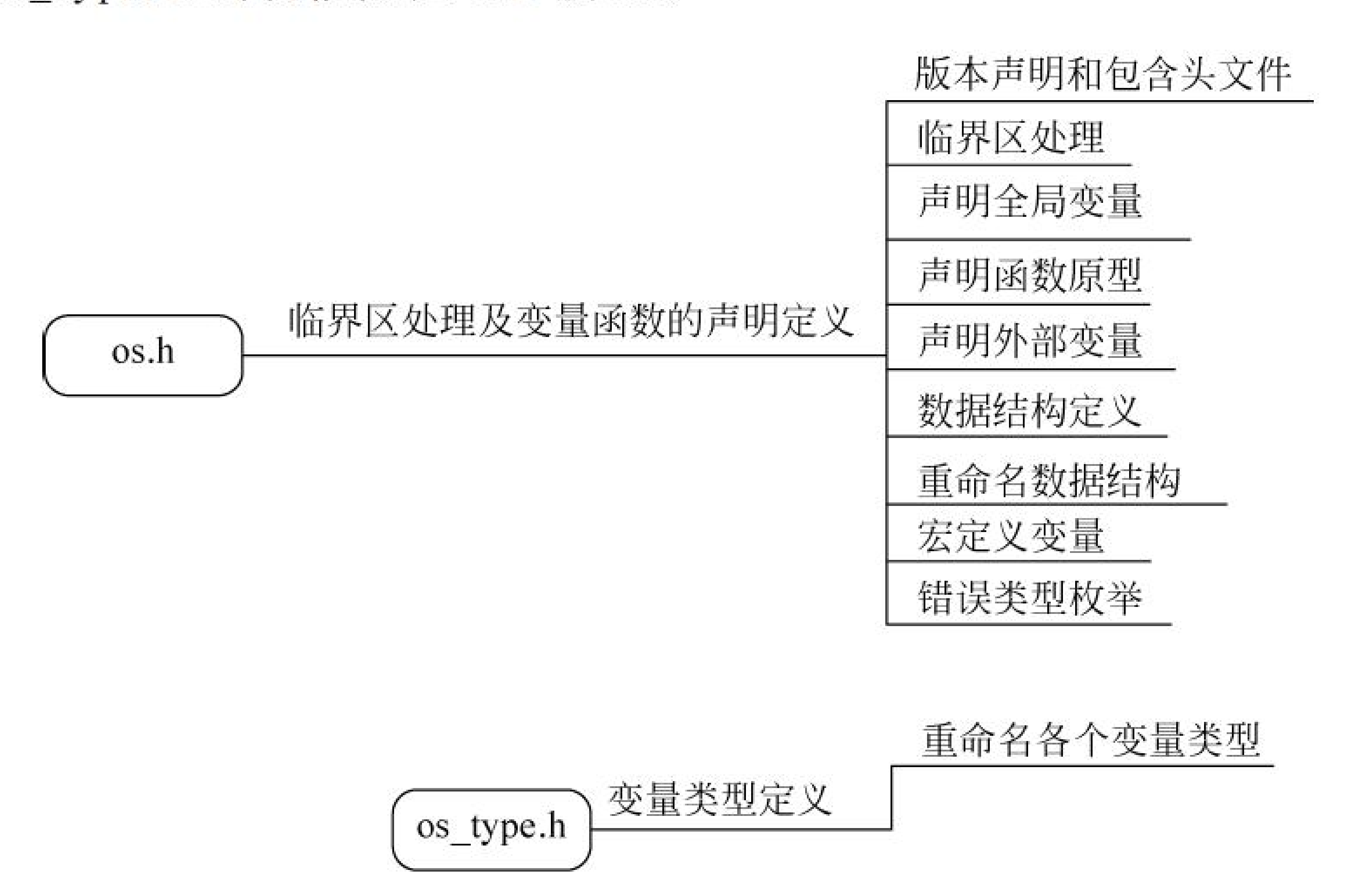


图 1.2 os.h 和 os_type.h 功能概况

1.1.2 os_core.c 概况

os_core.c 作为 μC/OS-Ⅲ内核的核心，定义了系统的核心功能，如系统的初始化，任务调度操作，启动多任务处理等，具体功能如图 1.3 所示。

1.1.3 os_task.c、os_prio.c 和 os_pend_multi.c 概况

os_task.c、os_prio.c 和 os_pend_multi.c 的功能概况如图 1.4 所示。

os_task.c 是系统的任务处理模块，包括任务的创建、删除、挂起、状态查看等功能，μC/OS-Ⅲ中任务可使用系统所提供的大部分函数实现自己的功能。

os_pend_multi.c 负责阻塞多任务，提供获取就绪队列的方法，并能判别请求资源的数据类型是信号量还是队列。

os_prio.c 定义了优先级，包括优先级表的初始化，得到最高优先级，以及添加和删除优先级的方法。

图 1.3 os_core.c 功能概况

图 1.4 os_task.c、os_prio.c 和 os_pend_multi.c 功能概况

1.1.4 os_flag.c 概况

os_flag 是系统中管理事件标志的模块，它主要负责事件标志的创建和删除，设置和清除标志位等操作，如图 1.5 所示。

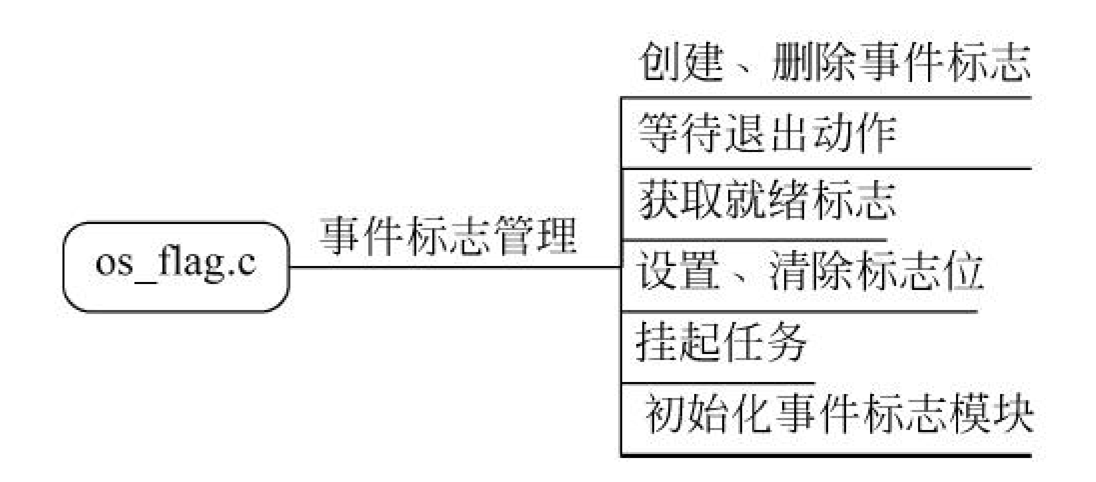


图 1.5 os_flag.c 功能概况

1.1.5 os_sem.c 和 os_mutex.c 概况

os_sem.c 和 os_mutex.c 分别提供了信号量和互斥锁的管理方法，包括创建、删除信号量和互斥锁，请求、释放信号量和互斥锁，以及设定、清除信号量和互斥锁等方法，为各个任务提供了同步和互斥关系的实现方案。具体功能如图 1.6 所示。

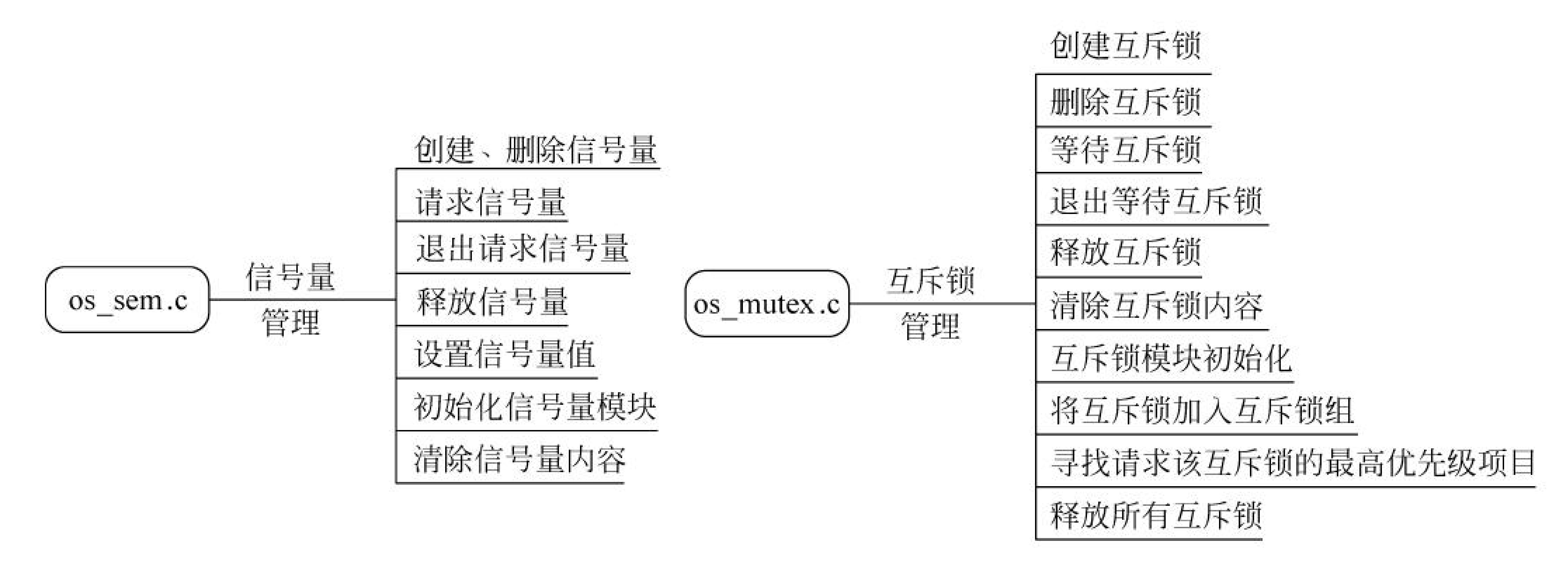


图 1.6 os_sem.c 和 os_mutex.c 功能概况

1.1.6 os_q.c 和 os_msg.c 概况

os_msg.c 和 os_q.c 分别提供了消息处理和消息队列管理功能，为任务间的消息传递提供了方法。中断服务程序或者任务都可以将消息发送到目标任务的消息队列，如果消息队列为空，目标任务将会被置入挂起队列。具体功能如图 1.7 所示。

1.1.7 os_tick.c、os_time.c 和 os_tmr.c 概况

μC/OS-Ⅲ中管理时间的源文件是 os_tmr.c、os_time.c 和 os_tick.c，它们分别提供了定时器管理，时间管理和时钟管理服务。定时器管理提供了定时器的使用方法，时间管理提

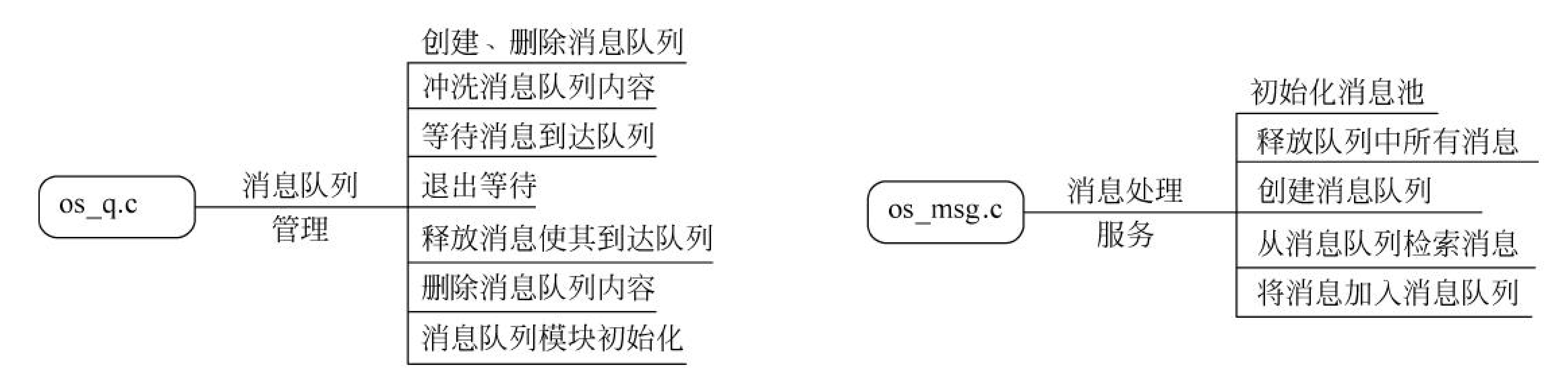


图 1.7 os_q.c 和 os_msg.c 功能概况

供了任务延时、获取系统时间等方法，而时钟管理则提供了时钟延时任务管理的服务。具体功能如图 1.8 所示。

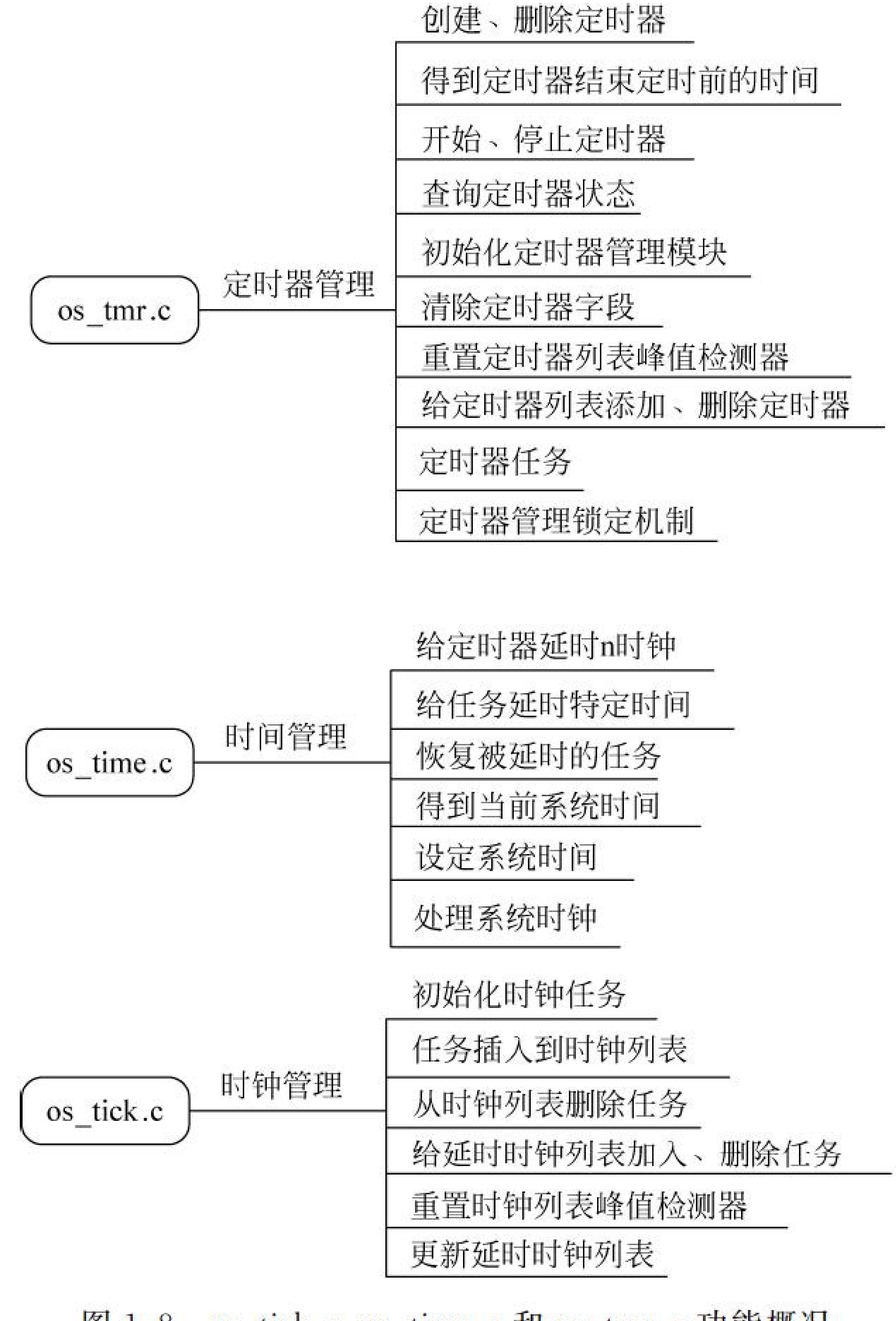


图 1.8 os_tick.c、os_time.c 和 os_tmr.c 功能概况

1.1.8 os_int.c 概况

os_int.c 提供了管理中断服务程序队列的方法。具体功能如图 1.9 所示。



图 1.9 os_int.c 功能概况

1.1.9 os_mem.c 概况

os_mem.c 是 μC/OS-Ⅲ中负责内存分区管理的模块，它负责创建内存分区，获取和释放内存块，为其他任务提供内存控制接口。内存管理功能具体如图 1.10 所示。

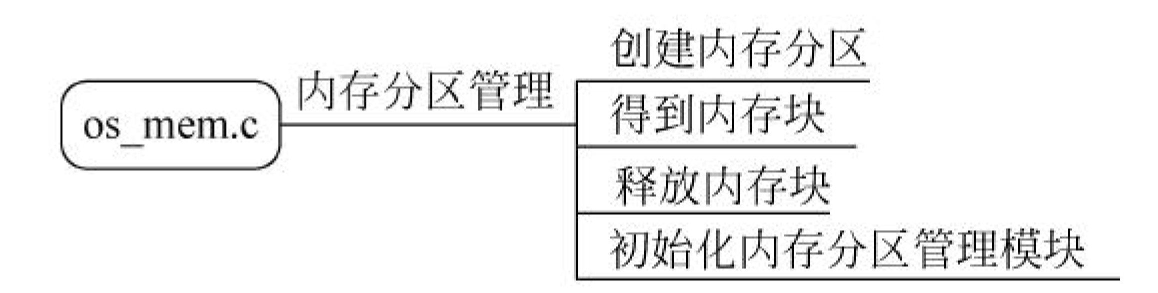


图 1.10 os_mem.c 功能概况

1.1.10 os_dbg.c、os_cfg_app.c 和 os_stat.c 概况

os_dbg.c 主要提供保存调试器所需常量的方法和初始化系统调试器的方法。

os_cfg_app.c 可对系统的配置进行初始化操作，并声明数据的存储方式。

os_stat.c 是 μC/OS-Ⅲ中对系统状态进行统计的模块，可计算 CPU 的负载等信息，提供有统计任务的管理和初始化方法。具体功能如图 1.11 所示。

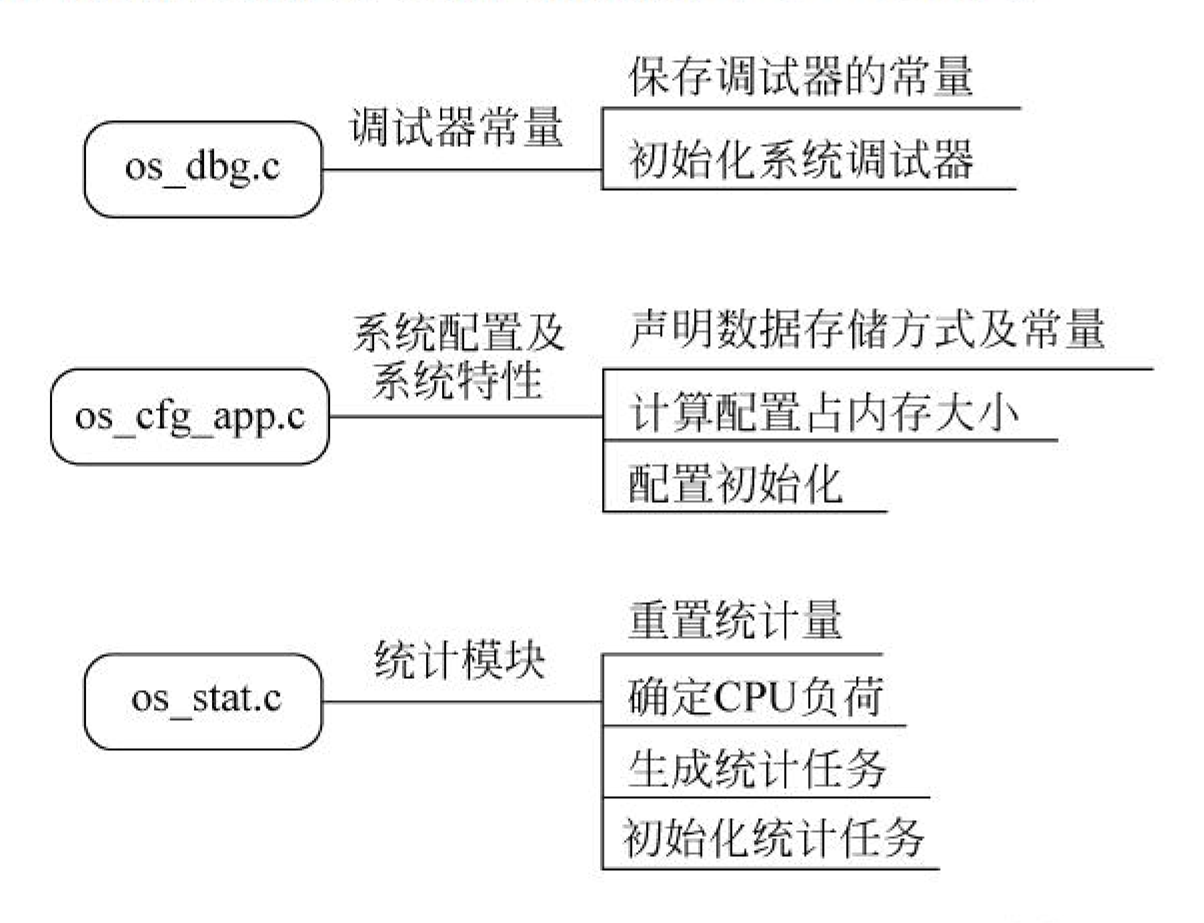


图 1.11 os_dbg.c、os_cfg_app.c 和 os_stat.c 功能概况

1.1.11 os_cfg.h 概况

os_cfg.h 文件是系统的编译配置文件，用来设置系统的功能选项。最常用的选项有：

```
#define OS_CFG_PRIO_MAX                32u    \\设置系统的最大优先级
#define OS_CFG_STK_SIZE_MIN            64u    \\设置任务栈的最小
#define OS_CFG_SCHED_ROUND_ROBIN_EN    1u     \\启用 RR 调度
#define OS_CFG_MEM_EN                  1u     \\启用内存管理
#define OS_CFG_FLAG_EN                 1u     \\启用事件标志
#define OS_CFG_MUTEX_EN                1u     \\启用互斥信号量
#define OS_CFG_Q_EN                    1u     \\启用消息队列
#define OS_CFG_SEM_EN                  1u     \\启用信号量
#define OS_CFG_STAT_TASK_EN            1u     \\启用统计任务
#define OS_CFG_TMR_EN                  1u     \\启用定时器
```

1.2 μC/OS-Ⅲ概览

μC/OS-Ⅲ是一个可扩展、可固化、抢占式的实时内核。它是第三代内核，提供了现代实时内核所期望的所有功能，包括资源管理、同步、内部任务交流、任务管理、时间管理、消息队列、软件定时器、内存分区等。μC/OS-Ⅲ也具有很多在其他实时内核中所没有的特性。比如能在运行时测量运行性能，直接发送信号或消息给任务，任务能同时等待多个信号量和消息队列等。下面将从几个方面具体阐述内核机制。

1.2.1 任务管理

μC/OS-Ⅲ对任务数量无限制。实际上，任务的数量受限于处理器能提供的内存大小。每一个任务需要有自己的堆栈空间，μC/OS-Ⅲ在运行时监控任务堆栈的生长。μC/OS-Ⅲ对任务的优先级数无限制，然而，配置 μC/OS-Ⅲ的优先级在 32 到 256 之间已经能满足大多数的应用需求了。

每个任务都有一个任务控制块(Task Control Block，TCB)，这是一个比较复杂的数据结构。在任务控制块偏移为 0 处，存储着一个指针，它记录了所属任务的专用堆栈地址。任务控制块是被 μC/OS-Ⅲ用于维护任务的一个结构体。每个任务都必须有自己的 TCB。μC/OS-Ⅲ在 RAM 中分配 TCB。当调用 μC/OS-Ⅲ提供的与任务相关的函数(以 OSTask???()形式命名)时，任务的 TCB 地址会被提供给该函数。TCB 的结构定义于 os.h 中。可以根据具体应用对 TCB 中的一些变量进行裁剪。用户程序不应该访问这些变量(尤其不能更改它们)，即 TCB 中的变量只能被 μC/OS-Ⅲ访问。

1. 任务创建：OSTaskCreate()，OSTaskCreateExt()

创建一个任务时必须为其分配一个 TCB、一个堆栈、一个优先级和其他一些参数。任务以无限循环的方式实现。另外，任务不允许有返回值。

2. 任务删除：OSTaskDel()，OSTaskDelReq()

任务的使命完成后，就要调用 OSTaskDel()删除该任务。OSTaskDel()实际上不是删除任务的代码，而是让任务不再具有使用 CPU 的资格。

3．任务挂起：OSTaskSuspend()

μC/OS-Ⅲ将等待信号量、mutex、事件标志组和消息队列的任务存储到挂起队列中。挂起队列中包括数据结构 OS_PEND_LIST。它包含在另一个叫做 OS_PEND_OBJ 的数据结构中。任务不是直接链接到挂起队列中，而是通过叫做 OS_PEND_DATA 的数据结构作为媒介进行链接。OS_PEND_DATA 是在任务被放入挂起队列中时分配到任务堆栈的。用户代码不能直接访问挂起队列，必须调用 μC/OS-Ⅲ所提供的函数。

μC/OS-Ⅲ中所有的挂起服务都可以有时间限制，预防死锁。值得注意的是，任务在等待事件时，不会占用 CPU。

4．系统内部任务

μC/OS-Ⅲ可创建五个内部任务：空闲任务，时基任务，中断处理任务，统计任务，定时器任务。空闲任务和时基任务是必需的，统计任务、定时器任务与中断处理任务是可选择的。

1.2.2 任务调度

μC/OS-Ⅲ是一个抢占式多任务处理内核。因此，正在运行的总是最重要的就绪任务。每个任务都需要被设定一个优先级。μC/OS-Ⅲ的工作是决定哪个任务应该占用 CPU。一般地，μC/OS-Ⅲ选择就绪队列中优先级最高的任务运行。在 μC/OS-Ⅲ中，数值越小优先级越高。μC/OS-Ⅲ允许用户在编译时配置优先级的范围(详见 os_cfg.h 文件中 OS_PRIO_MAX)。此外，μC/OS-Ⅲ允许任务具有相同优先级且对该任务数无限制。当多个相同优先级的任务就绪，且此优先级是目前最高时，系统使用时间片轮转法进行任务调度。

当 μC/OS-Ⅲ转向执行另一个任务时，会保存当前任务的 CPU 寄存器到堆栈，并将新任务堆栈中的 CPU 寄存器载入 CPU。这个过程叫做上下文切换。上下文切换需要一些开支。CPU 的寄存器越多，开支越大。上下文切换的时间基本取决于有多少个 CPU 寄存器需被存储和载入。在 μC/OS-Ⅲ中，任务切换时的堆栈设置类似于中断发生时的情况，所有的 CPU 寄存器都将被保存。假定任务堆栈中的信息将要被载入到 CPU 中，而任务堆栈指针(TSP)指向任务堆栈中最后一个被保存的寄存器。程序指针寄存器和状态寄存器最先被保存在任务堆栈中。中断堆栈指针(ISP)指向当前中断堆栈的顶部。当中断服务程序被执行时，处理器把 R14 作为堆栈指针用于指向函数和局部参数。

有任务级和中断级两种上下文切换方式。任务级切换通过调用 OSCtxSw()实现，实际上它是被宏 OS_TASK_SW()调用的。中断级切换通过调用 OSIntCtxSw()实现，它用汇编语言编写，保存在 OS_CPU_A.ASM 中。

从用户的观点来看，任务可以有 5 种状态。图 1.12 展示了休眠状态、就绪状态、运行状态、等待状态、中断状态 5 种任务状态间的转换关系。

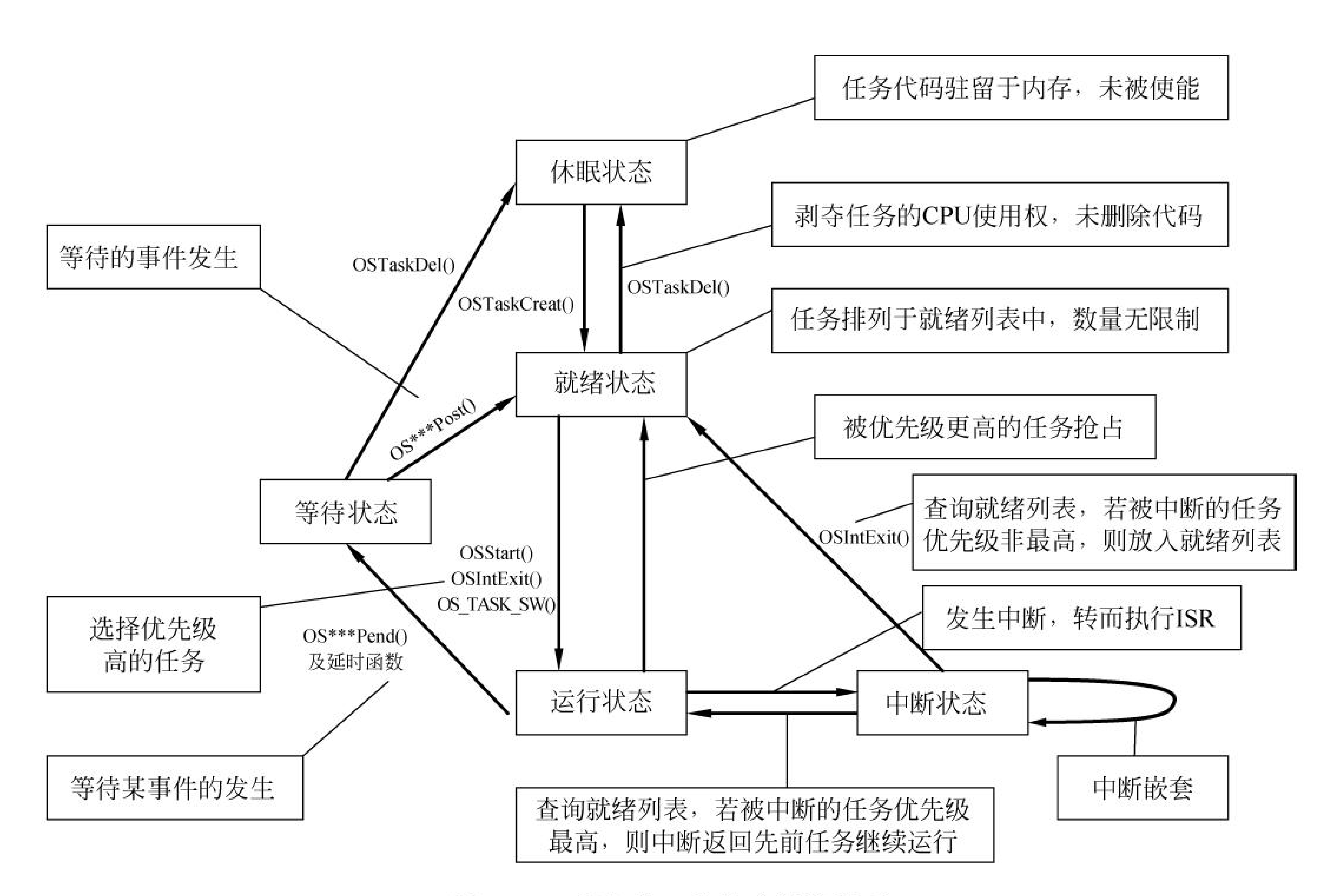


图 1.12　任务的 5 种状态转换关系

1.2.3　任务同步

μC/OS-Ⅲ中用于同步的机制有信号量和事件标志组两种。

1. 信号量

任务信号量：μC/OS-Ⅲ内核允许中断服务程序(ISR)或者任务直接发送信号量给其他任务。信号量可以标志事件的发生，也可以用于保护共享资源。

互斥信号量(mutex)：是一个内核对象(结构体)，用于保护共享资源。任务要访问共享资源就必须先获得 mutex。mutex 的占有者使用完这个资源后必须释放。

2. 事件标志组

当任务要与多个事件同步时可以使用事件标志。若其中的任意一个事件发生时任务就绪，叫做逻辑或(OR)。若所有的事件都发生时任务就绪，叫做逻辑与(AND)。

用户可以创建任意个事件标志组(受限于 RAM)。μC/OS-Ⅲ中与事件标志组相关的函数都使用 OSFlg***()。与事件标志组相关的函数代码都在 os_flag.c 中。设置 os_cfg.h 中的 OS_CFG_FLAG_EN 为 1，开启事件标志组功能。事件标志组是μC/OS-Ⅲ的内核对象，以 OS_FLAG_GRP 为数据类型。它可以是 8 位，16 位，32 位，决定于 os_type.h 中所定义的 S_FLAGS。事件标志组中的位是任务所等待事件是否发生的标志。事件标志组必须在创建后使用。任务或 ISR 可以提交标志。然而，只有任务可以将在事件标志组中等待的

其他任务删除或取消等待，也只有任务才能让任务在事件标志组中等待。任务可以等待事件标志组中的任意个位被设置。等待也可以被设置期限，以时基为单位。任务等待事件标志组中的位，可以被设置为 OR 模式，或者是 AND 模式。

1.2.4 任务间通信

任务或 ISR 与另一个任务进行通信的这种信息交换叫做作业间的通信。有全局变量和发送消息两种方法实现这种通信。

消息队列是一个内核对象，ISR 或任务可以直接发送消息到另一个任务。发送者指定一个消息并将其发送到目标任务的消息队列。目标任务等待消息的到达。消息到达后，目标任务取得这些消息。如果消息队列为空，目标将会被安放在挂起队列中并与消息队列保持联系。

消息队列是先入先出模式（FIFO）。μC/OS-Ⅲ也可以将其设置为后入先出模式（LIFO）。在任务或 ISR 发送紧急消息给另一个任务时，后入先出模式是非常有用的，在这种情况下，该紧急消息绕过消息队列中的其他消息。消息队列的长度可以在运行时设置。消息队列中存放了等待该消息的任务。多个任务可以在消息队列中等待消息。当一个消息被发送到消息队列时，等待该消息的高优先级任务接收这个消息。消息发送者可以广播这个消息给消息队列中的所有任务。在这种情况下，如果接收到的消息中有优先级高于消息发送者优先级的任务，μC/OS-Ⅲ就会切换到这个高优先级的任务。注意：不是每个任务都需要设置等待期限，有些任务可能需要永远等待这个消息。

1.2.5 中断

中断是硬件机制，用于通知 CPU 有异步事件发生。当中断被响应时，CPU 保存部分（或全部）寄存器值并跳转到中断服务程序（ISR）。ISR 响应这个事件，当 ISR 处理完成后，程序会返回中断前的任务或更高优先级的任务。

μC/OS-Ⅲ可以通过锁定调度器的方式来代替关中断，因此关中断的时间会非常少，这样就使 μC/OS-Ⅲ可以响应一些非常快的中断源了。

μC/OS-Ⅲ的中断响应时间是可确定的，μC/OS-Ⅲ提供的大部分服务的执行时间也是可确定的。若中断发生，中断会挂起正在执行的任务并去处理 ISR。ISR 中可能有某些任务等待的事件。一般来说，中断用来通知任务某些事件的发生，并在任务级处理实际的响应操作。ISR 程序越短越好，实际响应中断的操作应该被设置在任务级以便能让 μC/OS-Ⅲ管理这些操作。ISR 中只允许调用一些提交函数（OSFlagPost()，OSQPost()，OSSemPost()，OSTaskQPost()，OSTaskSemPost()），除了 OSMutexPost()。这是因为 mutex 只允许在任务级被修改。

一个中断被另一个中断所抢占的行为叫做中断嵌套，大多数处理器支持中断嵌套。然而，如果管理不当，中断嵌套很容易引起堆栈溢出。

1.2.6 时间管理

μC/OS-Ⅲ提供了软件定时器服务。当设置 os_cfg.h 中的 OS_CFG_TMR_EN 为 1 时,软件定时器服务被使能。定时器递减其计数值,当计数值为 0 时,即定时器期满。通过回调函数执行相应的操作(打开或关闭 LED、开启电机或其他操作)。回调函数是用户定义的,当定时器期满时可以被调用。

应用中可以定义任意个定时器(受限于处理器的 RAM)。μC/OS-Ⅲ的定时器服务通过调用 OSTmr???()开始。μC/OS-Ⅲ定时器的分辨率取决于时基频率,也就是变量 OS_CFG_TMR_TASK_RATE_HZ 的值,它以 Hz 为单位。如果时基任务的频率设置为 10Hz,则所有定时器的分辨率为十分之一秒。事实上,这是定时器的推荐值。定时器用于不精确时间尺度的任务。

μC/OS-Ⅲ需要一个能提供周期性时间的时基源,即系统时基。硬件定时器可以设置为每秒产生 10 次到 1000 次的中断来提供系统时基,也可以从交流电中获得 50Hz 到 60Hz 的时基源。事实上,也可以从交流电中获得 100Hz 到 120Hz 的过零点作为时基。时基可以看做是系统的心跳。它的速率取决于时基源。时基速率越快,系统的额外支出就越大。

μC/OS-Ⅲ需要时基源,用于任务的延时和超时功能。

(1) 如果 μC/OS-Ⅲ设置为延迟提交方式,μC/OS-Ⅲ读取当前时间戳并放到中断处理队列。然后中断队列处理任务发送信号量给时基任务。

(2) 如果 μC/OS-Ⅲ设置为直接提交方式,时基 ISR 直接发送信号量给时基任务。

1.2.7 内存管理

用户可以创建任意个内存分区(受限于处理器的 RAM)。μC/OS-Ⅲ中与内存分区相关的函数都使用 OSMem *** ()的形式。通过设置 os_cfg.h 中的 OS_CFG_MEM_EN 为 1 开启内存管理服务。OSMemGet()和 OSMemPut()可以在 ISR 中调用。

调用 OSMemCreate()创建一个内存分区。

(1) 当创建一个内存分区时,内存控制块(OS_MEM)的地址需作为参数。通常从静态内存中分配内存控制块,也可以调用 malloc()从堆中获得。用户代码不能释放内存控制块。

(2) OSMemCreate()将内存块链接起来并将其链表的首地址赋值给 OS_MEM 结构体。

(3) 每个内存块的大小必须大于保存一个指针变量所用的空间。

1.2.8 错误检测

μC/OS-Ⅲ能检测指针是否为 NULL,在 ISR 中调用的任务级服务是否允许,参数是否在允许范围内,配置选项的有效性,函数的执行结果等。每一个 μC/OS-Ⅲ的 API 函数返回

一个对应于函数调用结果的错误代号。

1.2.9 性能测量

μC/OS-Ⅲ有内置性能测量功能，能测量每一个任务的执行时间，每个任务的堆栈使用情况，任务的执行次数，CPU 的使用情况，ISR 到任务的切换时间，任务到任务的切换时间，列表中的峰值数，关中断和锁调度器平均时间等。

1.3 总体数据结构关系及描述

1.3.1 就绪任务管理

系统定义了长度为 OS_CFG_PRIO_MAX 的一维数组 OSRdyList，每个 OSRdyList 中含有头指针和尾指针，分别指向该优先级任务就绪双向链表的队首和队尾，如图 1.13 所示。

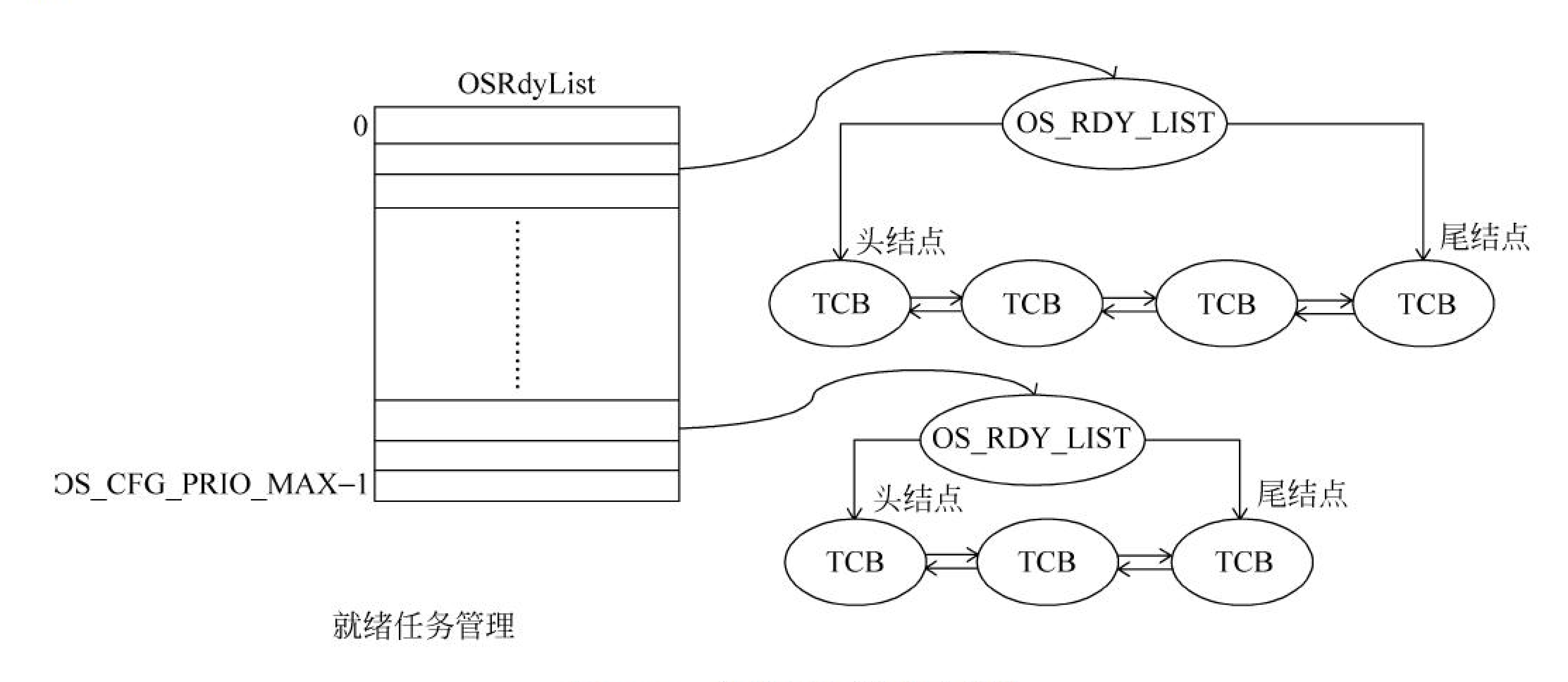


图 1.13 就绪任务管理示意图

1.3.2 事件标志和请求管理

系统定义了 os_flag_grp 结构体，它指向事件标志请求列表 os_pend_list，该列表包含指向请求详细信息双向链表头尾结点的指针，每个结点(os_pend_data)关联一个发出请求的任务信息(TCB)，如图 1.14 所示。

1.3.3 消息队列管理

消息池(os_msg_pool)指向消息(os_msg)链表的头节点。os_msg_q 是一个供 os_msg 先进先出的队列，它具有指向队列首节点和尾节点的指针，如图 1.15 所示。

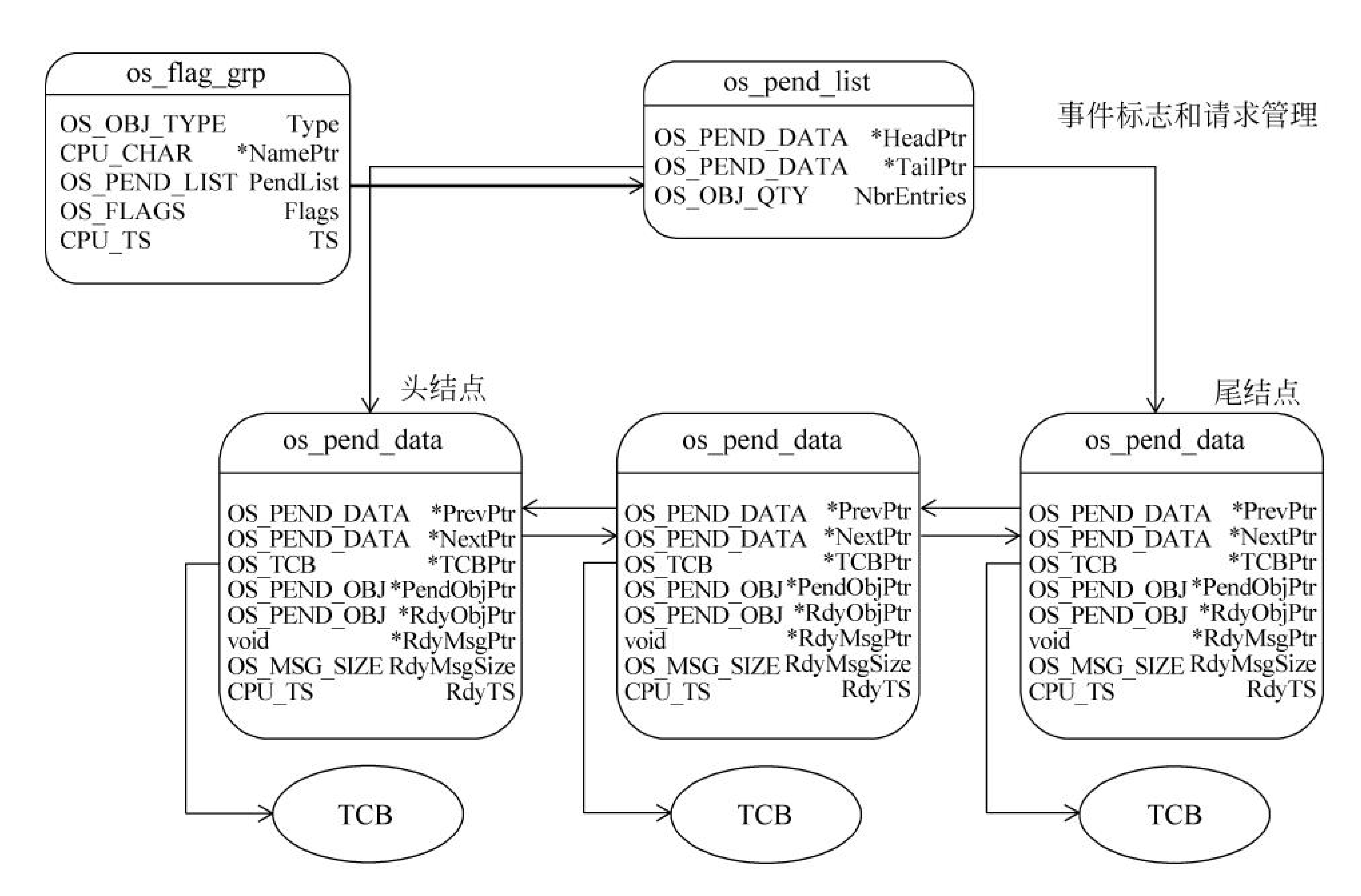


图 1.14 事件标志和请求管理示意图

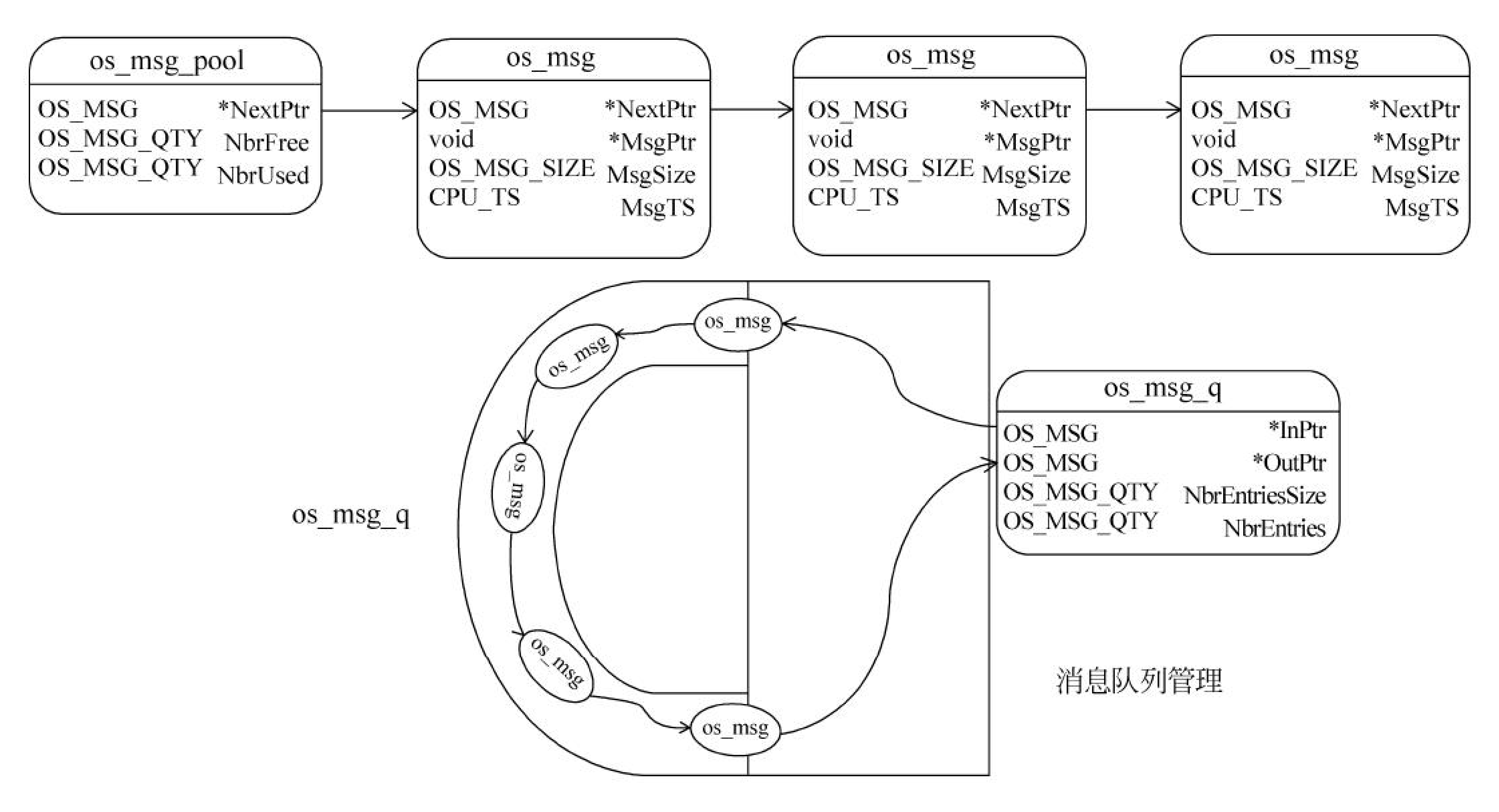


图 1.15 消息队列管理示意图

1.3.4 互斥信号量管理

系统定义了互斥信号量 os_mutex，它包含请求它的请求列表，并指向下一个 os_mutex，每个 os_mutex 与它所属的任务(TCB)相关联，如图 1.16 所示。

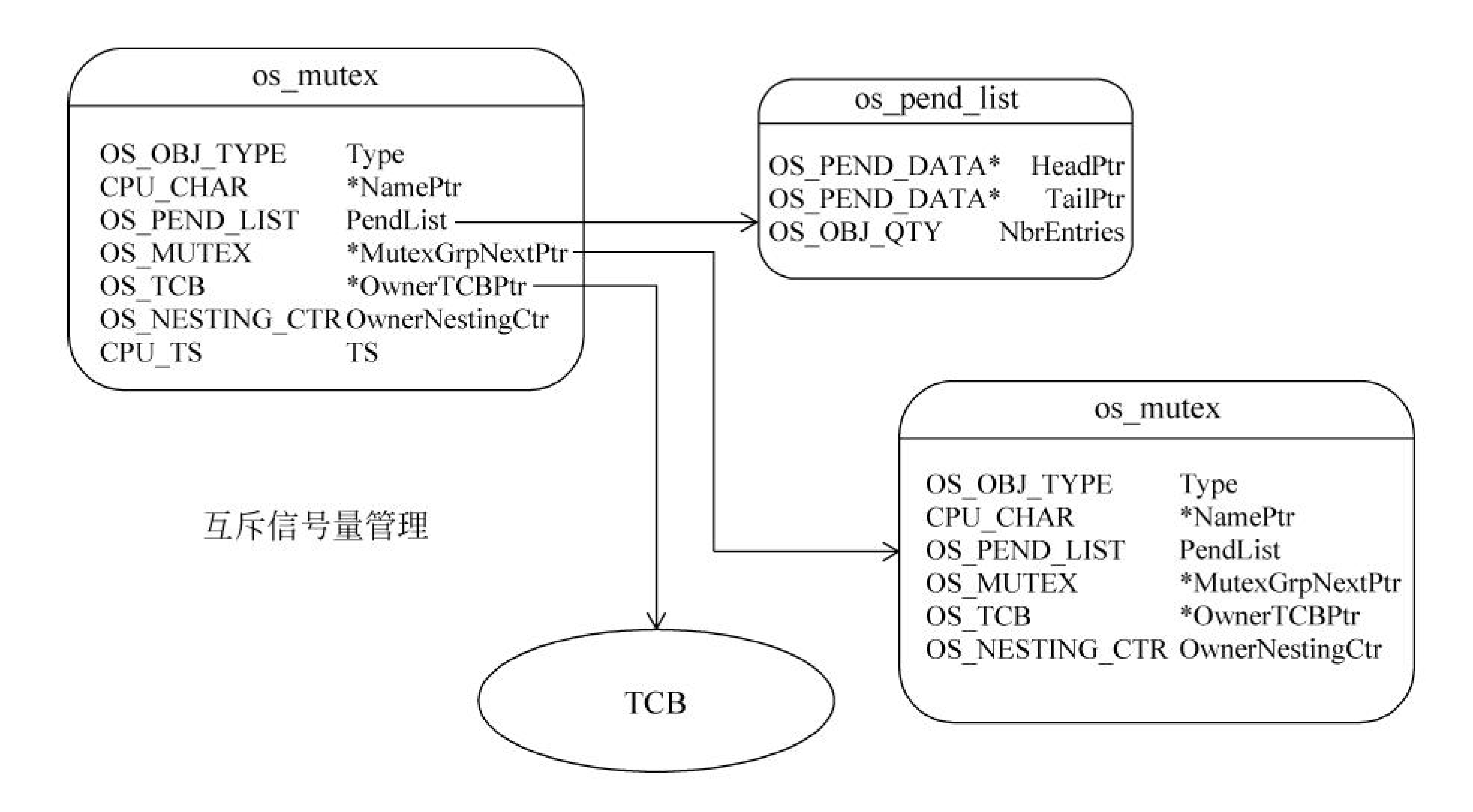


图 1.16 互斥信号量管理示意图

1.3.5 内存分区管理

系统定义了 os_mem 管理内存分区。它包含指向它所代表的内存分区的首地址，以及该分区中空闲块的首地址。每个块的起始 4 字节包含有指向下一块地址的指针，分区最后一个块的该位置写入 0 代表分区结束，如图 1.17 所示。

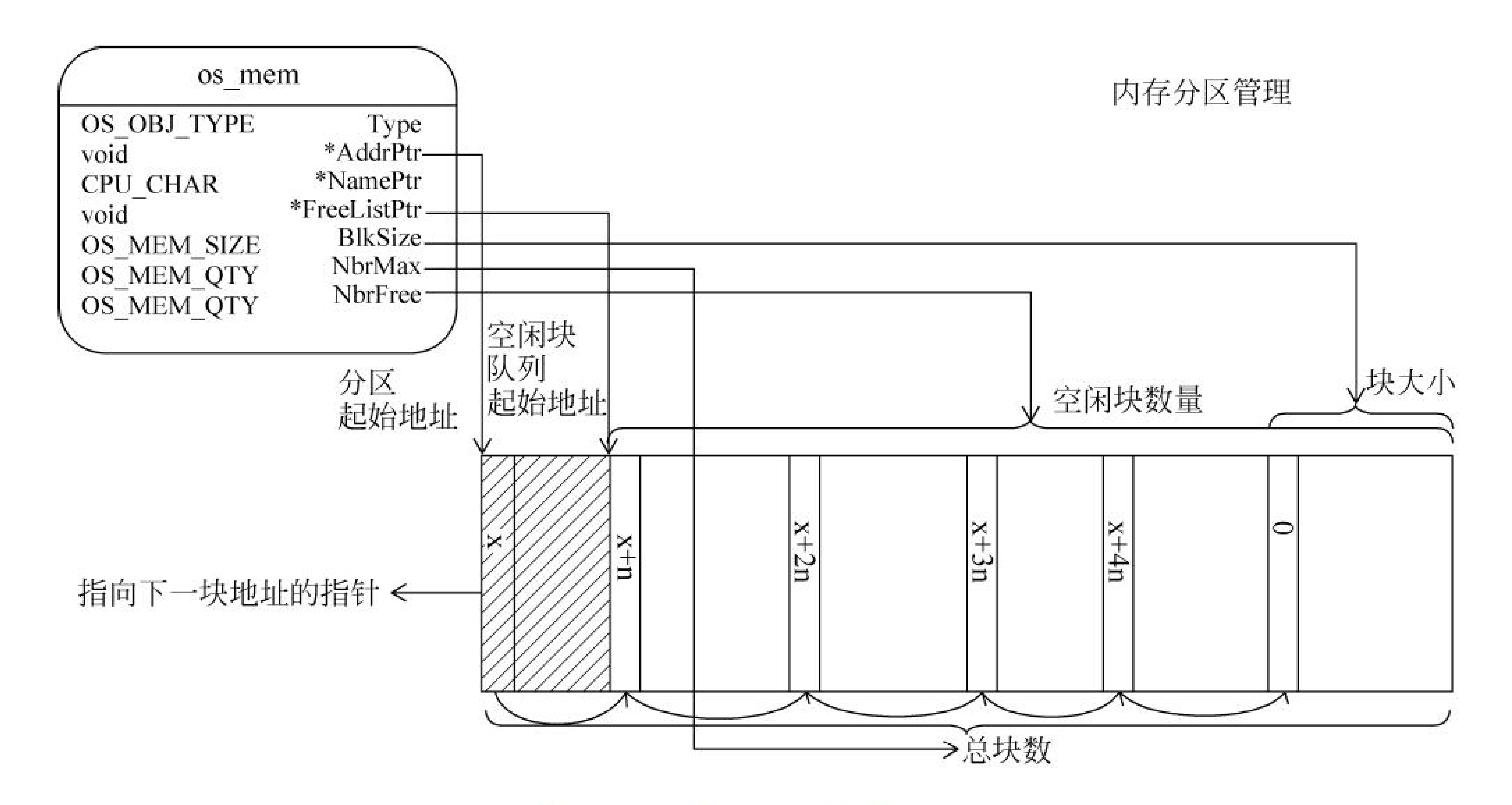


图 1.17 内存分区管理示意图

1.4 各关键数据结构描述

1.4.1 os_mem 成员定义

os_mem 结构定义如图 1.18 所示。

os_mem		
OS_OBJ_TYPE	Type	//类型
void	*AddrPtr	//分区初始地址
CPU_CHAR	*NamePtr	//分区名
void	*FreeListPtr	//空闲分区首地址
OS_MEM_SIZE	BlkSize	//块大小
OS_MEM_QTY	NbrMax	//总块数
OS_MEM_QTY	NbrFree	//空闲块数

图 1.18 os_mem 结构定义

1.4.2 os_flag_grp 成员定义

os_flag_grp 结构定义如图 1.19 所示。

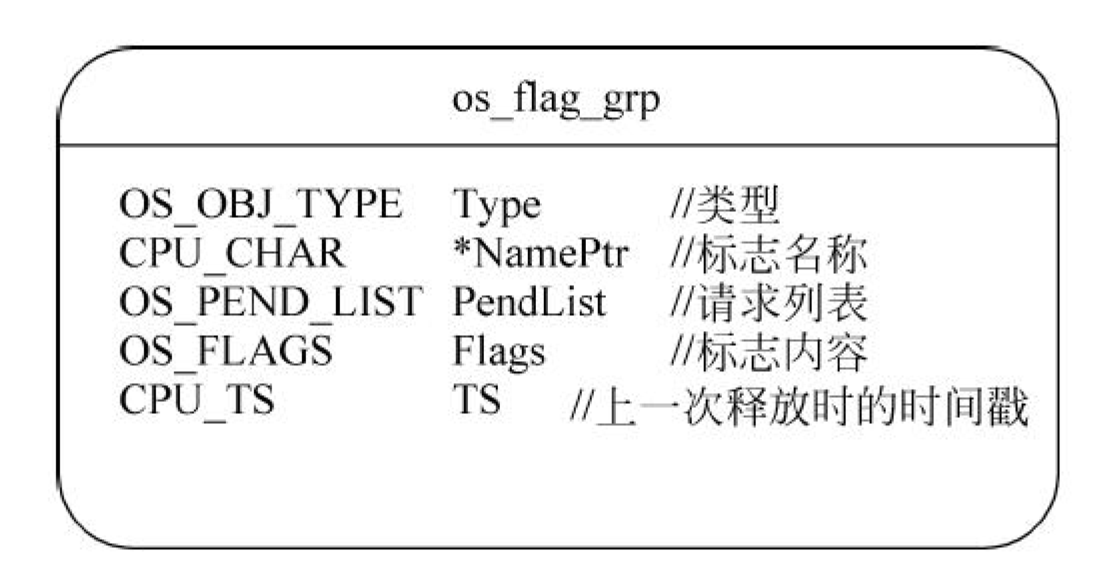


图 1.19 os_flag_grp 结构定义

1.4.3 OSPrioTbl 结构

优先级表(OSPrioTbl)是一个一维数组,数组中多个元素的每一位表示一个优先级,通过数 0 得到最高优先级,如图 1.20 所示。

1.4.4 os_mutex 成员定义

os_mutex 结构定义如图 1.21 所示。

1.4.5 os_tcb 成员定义

os_tcb 结构定义如图 1.22 所示。

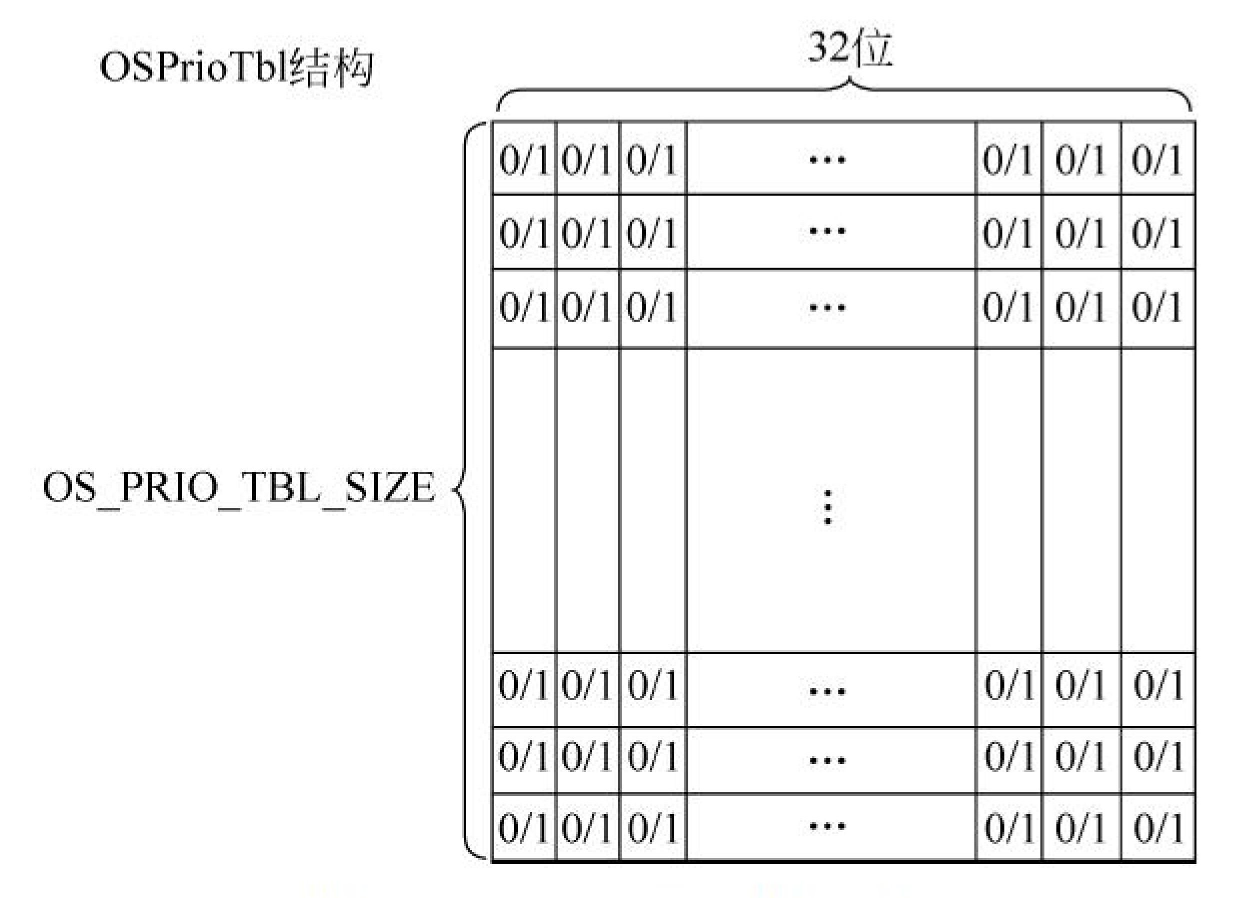


图 1.20 OSPrioTbl 结构定义

os_mutex		
OS_OBJ_TYPE	Type	//类型
CPU_CJAR	*NamePtr	//互斥信号量名称
OS_PEND_LIST	PendList	//请求列表
OS_MUTEX	*MutexGrpNextPtr	//下一个互斥信号量
OS_TCB	*OwnerTCBPtr	//所属的任务
OS_NESTING_CTR	OwnerNestingCtr	//互斥信号量值
CPU_TS	TS	//时间戳

图 1.21 os_mutex 结构定义

os_tcb		
CPU_STK	*StrPtr	//任务堆栈地址
CPU_CHAR	*NamePtr	//任务名称
OS_TCB	*NextPtr	//任务列表下一任务
OS_TCB	*PrevPtr	//任务列表上一任务
OS_TCB	*TickNextPtr	//时钟队列下一任务
OS_TCB	*TickPrevPtr	//时钟队列上一任务
OS_TICK_LIST	*TickListPtr	//时钟队列
OS_PEND_DATA	*PendDataTblPtr	//请求项目
OS_STATUS	PendStatus	//请求状态
OS_STATE	TaskState	//任务状态
OS_PRIO	Prio	//任务优先级
OS_MUTEX	*MutexGrpHeadPtr	//互斥信号量组头指针
OS_OPT	Opt	//选项
OS_TICK	TickRemain	//剩余时钟
void	*MsgPtr	//消息指针
OS_MSG_Q	MsgQ	//消息队列
OS_REG	RegTbl[OS_CFG_TASK_REG_TBL_SIZE]	//任务使用的寄存器
OS_FLAGS	FlagsPend	//等待事件标记
OS_FLAGS	FlagsRdy	//就绪事件标记
OS_NESTING_CTR	SuspendCtr	//嵌套次数

图 1.22 os_tcb 结构定义

1.5 内核函数

1.5.1 内核函数介绍

优先级相关函数如下：

(1) OS_PrioGetHighest()　查找最高优先级；

(2) OS_PrioInsert()　设置位映像表中相应的位；

(3) OS_PrioRemove()　清除位映像表中相应的位。

就绪列表相关的操作函数如下：

(1) OS_RdyListInit()　初始化就绪列表为空；

(2) OS_RdyListInsert()　插入一个 TCB 到就绪列表；

(3) OS_RdyListInsertHead()　插入一个 TCB 到就绪列表的头部；

(4) OS_RdyListInsertTail()　插入一个 TCB 到就绪列表的尾部；

(5) OS_RdyListMoveHeadToTail()　将 TCB 从列表头部移到尾部；

(6) OS_RdyListRemove()　将 TCB 从就绪列表中移除。

任务管理相关函数如下：

(1) OSTaskCreate(),OSTaskCreateExt()　任务创建；

(2) OSTaskDel(),OSTaskDelReq()　任务删除；

(3) OSTaskSuspend()　任务挂起；

(4) OSTaskResume()　任务恢复。

任务调度相关函数如下：

(1) OSIntCtxSw()　中断级任务切换；

(2) OSCtxSw()　任务级切换；

(3) OSSchedRoundRobinYield()　任务放弃分配给它的时间片；

(4) OSSched()　调度发生；

(5) OSIntExit()　退出中断服务程序时,调度发生函数；

(6) OSSchedUnlock()　调度器被解锁；

(7) OSSchedLock()　锁调度器。

定时器相关函数如下：

(1) OSTmrCreate()　创建和设置定时器；

(2) OSTmrDel()　删除一个定时器；

(3) OSTmrRemainGet()　获得定时器的剩余期限值；

(4) OSTmrStart()　开始定时器运行；

(5) OSTmrStateGet()　获得定时器当前状态；

(6) OSTmrStop()　暂停定时器。

时间服务相关函数如下：

(1) OSTimeDly() 延迟当前任务的执行；

(2) OSTimeDlyHMSM() 延时 HH:MM:SS. mmm 执行任务；

(3) OSTimeDlyResume() 恢复处于延时状态的任务；

(4) OSTimeGet() 获得当前的时基计数值；

(5) OSTimeSet() 设置当前的时基计数值；

(6) OSTimeTick() 用于标记，表示产生了一个时基中断。

任务信号量相关函数如下：

(1) OSSemCreate() 创建一个信号量；

(2) OSSemDel() 删除一个信号量；

(3) OSSemPend() 等待某个信号量；

(4) OSSemPendAbort() 取消等待某个信号量；

(5) OSSemPost() 释放或标记信号量；

(6) OSSemSet() 设置信号量计数值。

mutex 相关函数如下：

(1) OSMutexCreate() 创建一个 mutex；

(2) OSMutexDel() 删除一个 mutex；

(3) OSMutexPend() 等待一个 mutex；

(4) OSMutexPendAbort() 任务取消等待 mutex；

(5) OSMutexPost() 释放 mutex。

任务内建信号量相关函数如下：

(1) OSTaskSemPend() 等待一个任务信号量；

(2) OSTaskSemPendAbort() 取消等待；

(3) OSTaskSemPost() 发送信号量给任务；

(4) OSTaskSemSet() 设置信号量计数值。

事件标志组相关函数如下：

(1) OSFlagCreate() 创建一个事件标志组；

(2) OSFlagDel() 删除一个事件标志组；

(3) OSFlagPend() 在事件标志组中挂起；

(4) OSFlagPendAbort() 取消等待；

(5) OSFlagPendGetFalgsRdy() 获得事件标志组中任务就绪位；

(6) OSFlagPost() 提交标志到事件标志组。

消息队列相关函数如下：

(1) OSTaskQPend() 等待一个消息；

(2) OSTaskQPendAbort() 任务不再等待该消息；

(3) OSTaskQPost() 发送一个消息给任务；

(4) OSTaskQFlush()　清空这个消息队列。

内存分区相关函数如下：

(1) OSMemCreate()　创建一个内存分区；

(2) OSMemGet()　从内存分区中获得一个内存块；

(3) OSMemPut()　归还一个内存块给相应的内存分区。

1.5.2　关键代码分析

1. 任务切换函数 OSSched()

由以下函数源码可以得知，如果在中断中或调度器被锁住，则不能进行任务调度。如果在中断中进行任务调度，不仅会影响系统的实时性，还可能产生异常。正常情况下，系统总是运行优先级最高的任务。值得注意的是，由于 μC/OS-Ⅲ 允许同一优先级可以有多个任务，代码中还包含取同一优先级下第一个任务的语句。

```
void OSSched(void)
{
    CPU_SR_ALLOC();
    //若仍在中断中则不能进行任务调度
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        return;
    }
    //若调度器上锁则不能进行任务调度
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
        return;
    }
    //关中断
    CPU_INT_DIS();
    //找到就绪列表中优先级最高的任务的优先级
    OSPrioHighRdy  =  OS_PrioGetHighest();
    //找到最高优先级下的第一个任务
    OSTCBHighRdyPtr  =  OSRdyList[OSPrioHighRdy].HeadPtr;
    //判断该任务是否为当前任务,是就不用进行切换
    if (OSTCBHighRdyPtr  ==  OSTCBCurPtr) {
        CPU_INT_EN();
        return;
    }

#if OS_CFG_TASK_PROFILE_EN > 0u
    //被切换的任务次数加 1
    OSTCBHighRdyPtr->CtxSwCtr++;
#endif
    //总的任务切换次数加 1
```

```
    OSTaskCtxSwCtr++;

#if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
    OS_TLS_TaskSw();
#endif
    //调用任务切换的宏
    OS_TASK_SW();
    //使能中断
    CPU_INT_EN();

#ifdef OS_TASK_SW_SYNC
    OS_TASK_SW_SYNC();
#endif
}
```

2. 时间片轮转调度函数 OS_SchedRoundRobin()

若同一优先级中有多个任务,这些任务间则使用时间片轮转调度的方法进行切换。OS_SchedRoundRobin()就是执行这种操作的,它被 OSTimeTick()或者 OS_IntQTask()调用。当选择直接提交方式时,OS_SchedRoundRobin()被 OSTimeTick()调用。当选择延迟提交方式时,OS_SchedRoundRobin()被 OS_IntQTask()调用。

```
//是否包含时间片轮转代码
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
void  OS_SchedRoundRobin(OS_RDY_LIST   *p_rdy_list)
{
    OS_TCB     *p_tcb;
    //分配保存中断状态的局部变量
    CPU_SR_ALLOC();
    //若已使能时间片轮转调度允许标志则返回
    if (OSSchedRoundRobinEn != DEF_TRUE) {
        return;
    }
    //进入临界段,关中断
    CPU_CRITICAL_ENTER();
    //找到优先级列表中第一个任务 TCB
    p_tcb = p_rdy_list->HeadPtr;
    //若为空指针则退出
    if (p_tcb == (OS_TCB *)0) {
        CPU_CRITICAL_EXIT();
        return;
    }
    //若为空闲任务则不支持
```

```
        if (p_tcb == &OSIdleTaskTCB) {
            CPU_CRITICAL_EXIT();
            return;
        }
        //检查任务时间片是否大于 0
        if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {
            p_tcb->TimeQuantaCtr--;
        }
        //任务时间片还没用完,无须调度
        if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {
            CPU_CRITICAL_EXIT();
            return;
        }
        //若该优先级上只有一个任务则无须调度
        if (p_rdy_list->NbrEntries < (OS_OBJ_QTY)2) {
            CPU_CRITICAL_EXIT();
            return;
        }
        //若调度器被锁则无法调度
        if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
            CPU_CRITICAL_EXIT();
            return;
        }
        //任务时间片用完,进行任务切换
        OS_RdyListMoveHeadToTail(p_rdy_list);
        p_tcb = p_rdy_list->HeadPtr;
        //若未设置时间片大小则使用默认时间片值
        if (p_tcb->TimeQuanta == (OS_TICK)0) {
            p_tcb->TimeQuantaCtr = OSSchedRoundRobinDfltTimeQuanta;
        } else {
            p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;
        }
        //退出临界区,开中断
        CPU_CRITICAL_EXIT();
    }
    #endif
```

3. 中断嵌套管理

1) 进入中断 OSIntEnter()

函数首先检测系统是否在运行。若系统还未运行就进入中断,则因为中断返回任务将进行任务切换,系统还未运行就进行任务切换将会导致系统崩溃。所以中断初始化放在首个任务开始执行的地方,并且由函数源码可以看出,允许中断嵌套的层数上限为 250。

```
void  OSIntEnter(void)
{
    //确认系统正在运行
    if (OSRunning != OS_STATE_OS_RUNNING) {
        return;
    }
    //如果嵌套 250 层之后就退出
    if (OSIntNestingCtr >= (OS_NESTING_CTR)250u) {
        return;
    }
    //嵌套深度加 1
    OSIntNestingCtr++;
}
```

2）退出中断 OSIntExit()

该函数仍然先判断系统是否在运行，然后关中断进入临界段。若处在中断嵌套中，则返回到上级中断，否则返回执行就绪列表中优先级最高的任务。

```
void  OSIntExit(void)
{
    //分配保存中断的局部变量,使之在关中断时保持中断状态
    CPU_SR_ALLOC();
    if (OSRunning != OS_STATE_OS_RUNNING) {
        return;
    }
    //关中断
    CPU_INT_DIS();
    if (OSIntNestingCtr == (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }
    //中断嵌套层次减 1
    OSIntNestingCtr--;
    //若仍有中断在进行则返回上一级中断
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }
    //检查调度器是否被锁住,否则需要切换为就绪表中最高优先级任务
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }
```

```
    //找到优先级最高的第一个任务
    OSPrioHighRdy    = OS_PrioGetHighest();
    OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
    if (OSTCBHighRdyPtr == OSTCBCurPtr) {
        CPU_INT_EN();
        return;
    }

#if OS_CFG_TASK_PROFILE_EN > 0u
      //任务切换次数加 1
      OSTCBHighRdyPtr->CtxSwCtr++;
#endif
    //总的任务切换次数加 1
    OSTaskCtxSwCtr++;

#if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
    OS_TLS_TaskSw();
#endif

    //调度任务切换宏
    OSIntCtxSw();
    //开中断
    CPU_INT_EN();
}
```

4. 任务同步

μC/OS-Ⅲ中用于同步的两种机制是信号量和事件标志组。

(1) 信号量工作原理如图 1.23 所示。

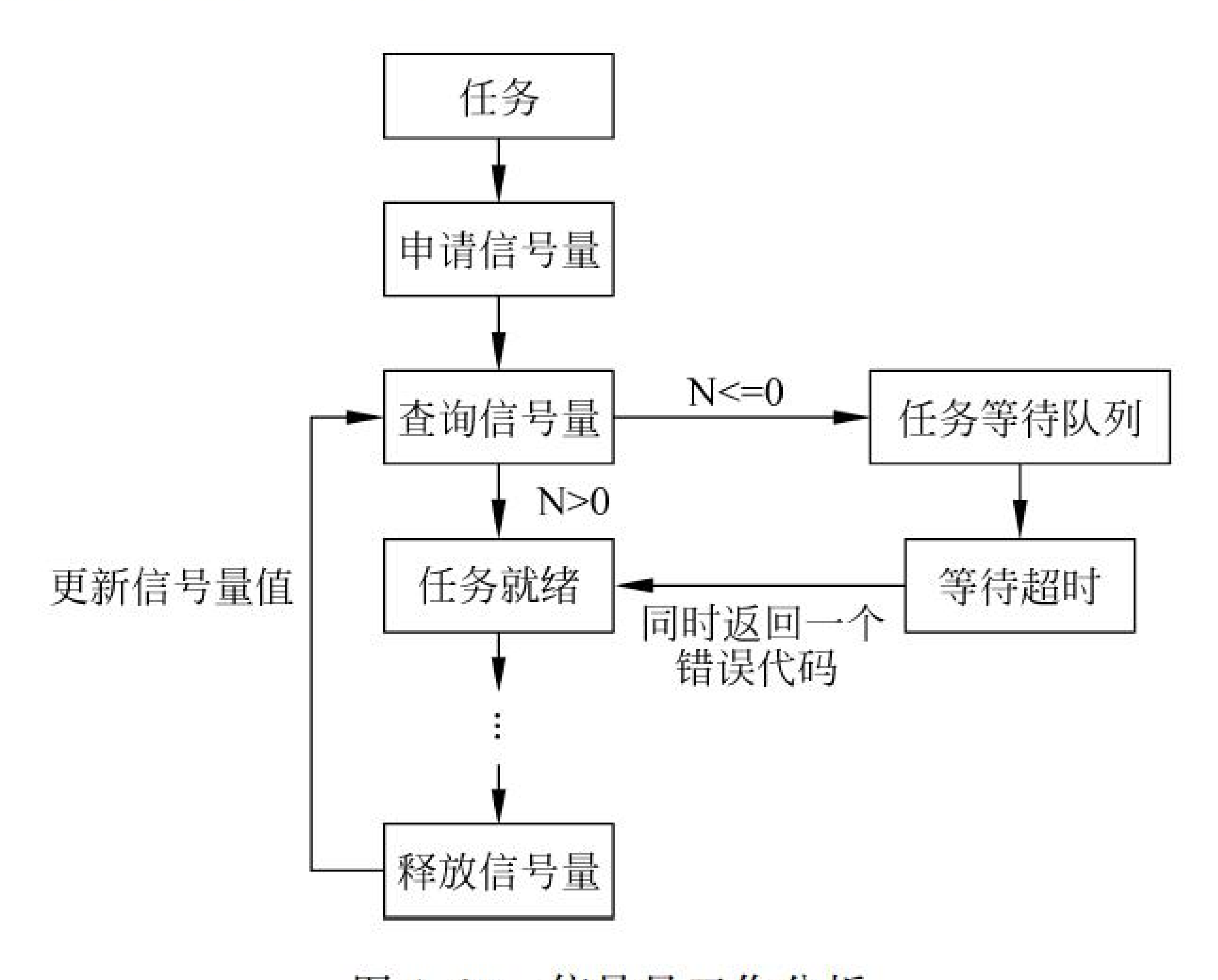


图 1.23 信号量工作分析

```
//任务信号量等待函数
OS_SEM_CTR  OSTaskSemPend (OS_TICK   timeout,
                           OS_OPT    opt,
                           CPU_TS    *p_ts,
                           OS_ERR    *p_err)
{
    OS_SEM_CTR    ctr;
    CPU_SR_ALLOC();
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
        TRACE_OS_TASK_SEM_PEND_FAILED(OSTCBCurPtr);
#endif
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_SEM_CTR)0);
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
        TRACE_OS_TASK_SEM_PEND_FAILED(OSTCBCurPtr);
#endif
       *p_err = OS_ERR_PEND_ISR;
        return ((OS_SEM_CTR)0);
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u
    switch (opt) {
        case OS_OPT_PEND_BLOCKING:
        case OS_OPT_PEND_NON_BLOCKING:
             break;
        default:
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
             TRACE_OS_TASK_SEM_PEND_FAILED(OSTCBCurPtr);
#endif
            *p_err = OS_ERR_OPT_INVALID;
             return ((OS_SEM_CTR)0);
    }
#endif

    if (p_ts != (CPU_TS *)0) {
       *p_ts  = (CPU_TS)0;
    }
    //进入临界区
```

```
    CPU_CRITICAL_ENTER();
    //若信号量值大于 0 表示信号量可用
    if (OSTCBCurPtr->SemCtr > (OS_SEM_CTR)0) {
        //任务信号量计数值减 1 后传给任务信号量计数器
      OSTCBCurPtr->SemCtr--;
        ctr    = OSTCBCurPtr->SemCtr;
        if (p_ts != (CPU_TS *)0) {
            *p_ts   = OSTCBCurPtr->TS;
        }
#if OS_CFG_TASK_PROFILE_EN > 0u
        //计算任务信号量从被提交到获取所用时间及最大时间
        OSTCBCurPtr->SemPendTime = OS_TS_GET() - OSTCBCurPtr->TS;
        if (OSTCBCurPtr->SemPendTimeMax < OSTCBCurPtr->SemPendTime) {
            OSTCBCurPtr->SemPendTimeMax = OSTCBCurPtr->SemPendTime;
        }
#endif
        CPU_CRITICAL_EXIT();
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
        TRACE_OS_TASK_SEM_PEND(OSTCBCurPtr);
#endif
       *p_err = OS_ERR_NONE;
        return (ctr);
    }
    //如果信号量不可用
    //根据可选项看是否进行等待
    if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) {
        CPU_CRITICAL_EXIT();
       *p_err = OS_ERR_PEND_WOULD_BLOCK;
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
        TRACE_OS_TASK_SEM_PEND_FAILED(OSTCBCurPtr);
#endif
        return ((OS_SEM_CTR)0);
    } else {
        //检查调度器是否被锁住,是则不等待
        if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
            CPU_CRITICAL_EXIT();
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
            TRACE_OS_TASK_SEM_PEND_FAILED(OSTCBCurPtr);
#endif
            *p_err = OS_ERR_SCHED_LOCKED;
            return ((OS_SEM_CTR)0);
        }
    }

    OS_CRITICAL_ENTER_CPU_EXIT();
    //将任务置于等待状态而不放入等待队列
```

```
    OS_Pend((OS_PEND_DATA  * )0,
            (OS_PEND_OBJ   * )0,
            (OS_STATE)OS_TASK_PEND_ON_TASK_SEM,
            (OS_TICK)timeout);
    OS_CRITICAL_EXIT_NO_SCHED();
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_OS_TASK_SEM_PEND_BLOCK(OSTCBCurPtr);
#endif
    //进行任务调度
    OSSched();
    CPU_CRITICAL_ENTER();
}
```

（2）事件标志组工作原理如图 1.24 所示。

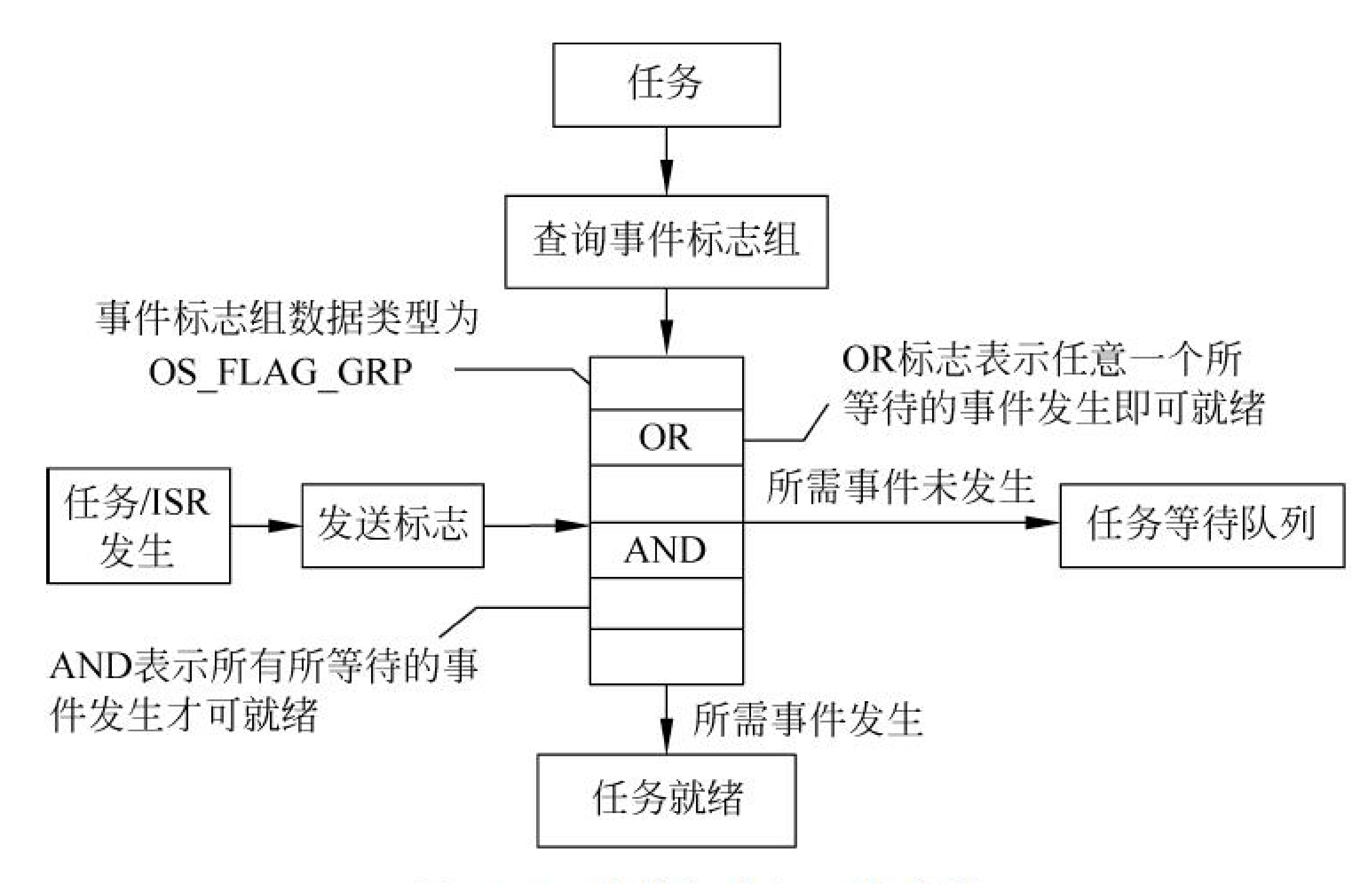


图 1.24　事件标志组工作分析

5. 内存分区创建函数 OSMemCreate()

创建一个内存分区，其工作原理如图 1.25 所示。

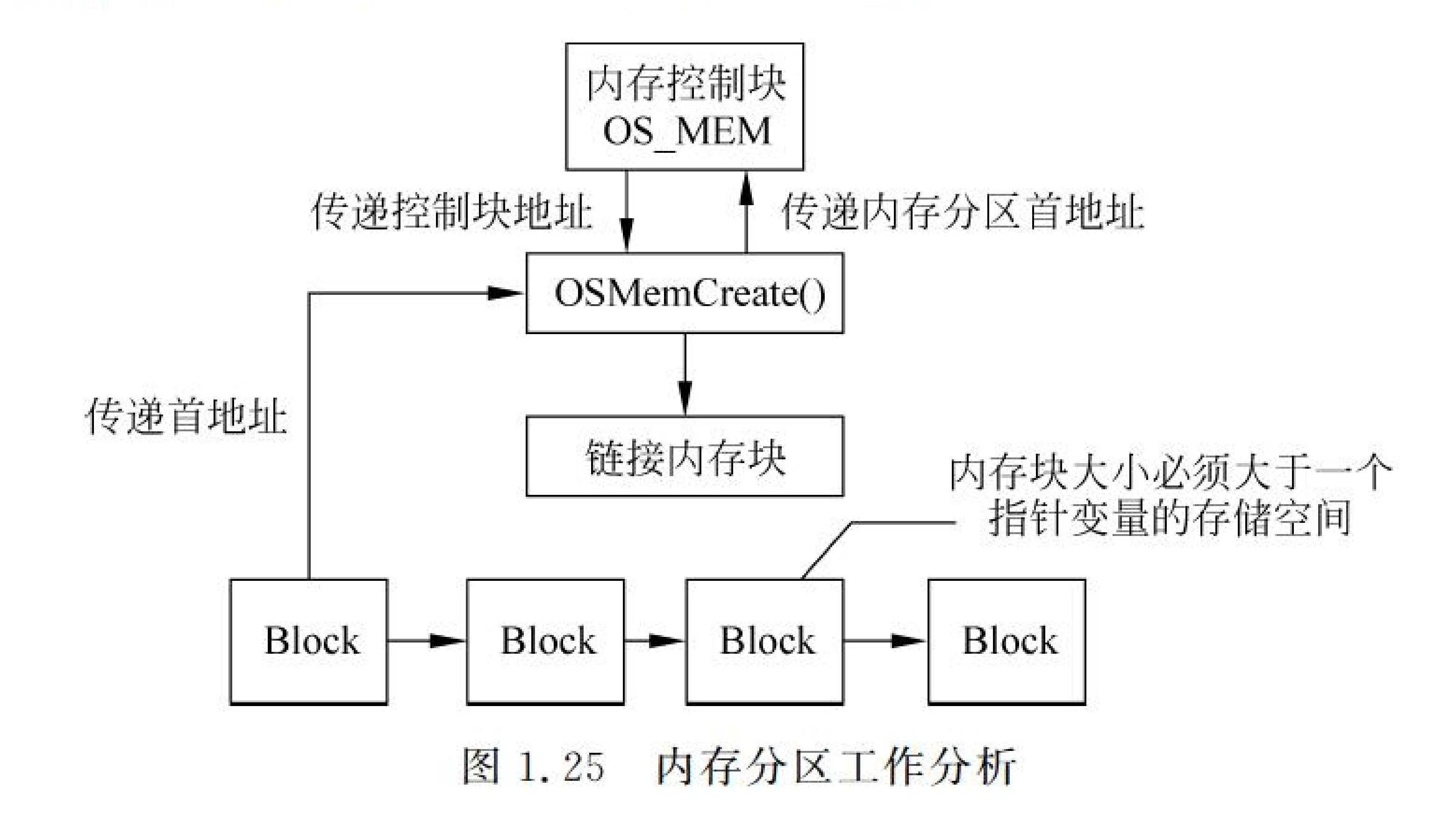


图 1.25　内存分区工作分析

习题

1. 什么是操作系统？嵌入式实时操作系统应该满足哪些最基本的要求？
2. 实时操作系统 μC/OS-Ⅲ主要由哪些功能模块构成？
3. 实时操作系统 μC/OS-Ⅲ的特点有哪些？
4. 可剥夺型内核和不可剥夺型内核的区别是什么？
5. 可重入函数与不可重入函数的区别是什么？

第 2 章



μC/OS-Ⅲ任务管理

2.1 μC/OS-Ⅲ任务管理机制

任务管理是嵌入式实时系统的核心和灵魂，决定了操作系统的实时性能。任务管理通常包括优先级设置、多任务调度机制和时间确定性等部分。嵌入式操作系统支持多任务，每个任务都具有优先级，任务越重要，赋予的优先级越高。优先级的设置分为静态优先级和动态优先级两种。静态优先级指的是每个任务在运行前都被赋予一个优先级，而且这个优先级在系统运行时不会改变；动态优先级则是指每个任务的优先级在系统运行时可以根据程序的需求动态改变。

μC/OS-Ⅲ对任务数量无限制。实际上，任务的数量受限于处理器可用的内存空间，包括任务代码存储空间和数据存储空间。每一个任务需要有自己的堆栈空间，μC/OS-Ⅲ在运行时监控任务堆栈的变化。μC/OS-Ⅲ对任务的优先级数无限制，然而，配置 μC/OS-Ⅲ的优先级在 32 到 256 之间已经能满足大多数的应用需求。

每个任务都有一个任务控制块(Task Control Block，TCB)，这是一个比较复杂的数据结构。在任务控制块偏移为 0 的地方，存储着一个指针，它记录了所属任务的专用堆栈地址。任务控制块是 μC/OS-Ⅲ用于维护任务的一个结构体。每个任务都必须有自己的 TCB。μC/OS-Ⅲ在 RAM 中分配 TCB，当调用 μC/OS-Ⅲ提供的与任务相关的函数时，任务的 TCB 地址会被提供给该函数。TCB 中的一些变量可以根据具体应用进行裁剪，但 TCB 变量只能被 μC/OS-Ⅲ访问。

TCB 的数据结构如下：

```
struct os_tcb {
    CPU_STK              * StkPtr;
    void                 * ExtPtr;
    CPU_STK              * StkLimitPtr;
    OS_TCB               * NextPtr;
    OS_TCB               * PrevPtr;
    OS_TCB               * TickNextPtr;
```

```
    OS_TCB              *TickPrevPtr;
    OS_TICK_LIST        *TickListPtr;
    CPU_CHAR            *NamePtr;
    CPU_STK             *StkBasePtr;
    OS_TLS              TLS_Tbl[OS_CFG_TLS_TBL_SIZE];
    OS_TASK_PTR         TaskEntryAddr;
    void                *TaskEntryArg;
    OS_PEND_DATA        *PendDataTblPtr;
    OS_STATE            PendOn;
    OS_STATUS           PendStatus;
    OS_STATE            TaskState;
    OS_PRIO             Prio;
    OS_PRIO             BasePrio;
    OS_MUTEX            *MutexGrpHeadPtr;
    CPU_STK_SIZE        StkSize;
    OS_OPT              Opt;
    OS_OBJ_QTY          PendDataTblEntries;
    CPU_TS              TS;
    CPU_INT08U          SemID;
    OS_SEM_CTR          SemCtr;
    OS_TICK             TickRemain;
    OS_TICK             TickCtrPrev;
    OS_TICK             TimeQuanta;
    OS_TICK             TimeQuantaCtr;
    void                *MsgPtr;
    OS_MSG_SIZE         MsgSize;
    OS_MSG_Q            MsgQ;
    CPU_TS              MsgQPendTime;
    CPU_TS              MsgQPendTimeMax;
    OS_REG              RegTbl[OS_CFG_TASK_REG_TBL_SIZE];
    OS_FLAGS            FlagsPend;
    OS_FLAGS            FlagsRdy;
    OS_OPT              FlagsOpt;
    OS_NESTING_CTR      SuspendCtr;
    OS_CPU_USAGE        CPUUsage;
    OS_CPU_USAGE        CPUUsageMax;
    OS_CTX_SW_CTR       CtxSwCtr;
    CPU_TS              CyclesDelta;
    CPU_TS              CyclesStart;
    OS_CYCLES           CyclesTotal;
    OS_CYCLES           CyclesTotalPrev;
    CPU_TS              SemPendTime;
    CPU_TS              SemPendTimeMax;
    CPU_STK_SIZE        StkUsed;
    CPU_STK_SIZE        StkFree;
    CPU_TS              IntDisTimeMax;
    CPU_TS              SchedLockTimeMax;
    OS_TCB              *DbgPrevPtr;
    OS_TCB              *DbgNextPtr;
    CPU_CHAR            *DbgNamePtr;
    CPU_INT08U          TaskID;
};
```

在 μC/OS-Ⅲ中,管理就绪任务的主要数据结构是 OSRdyList。每个 OSRdyList 中都包含头指针和尾指针,分别指向该优先级任务就绪双向链表的队首和队尾。OSRdyList 的本质是长度为 OS_CFG_PRIO_MAX 的一维数组,其基本结构如图 2.1 所示。

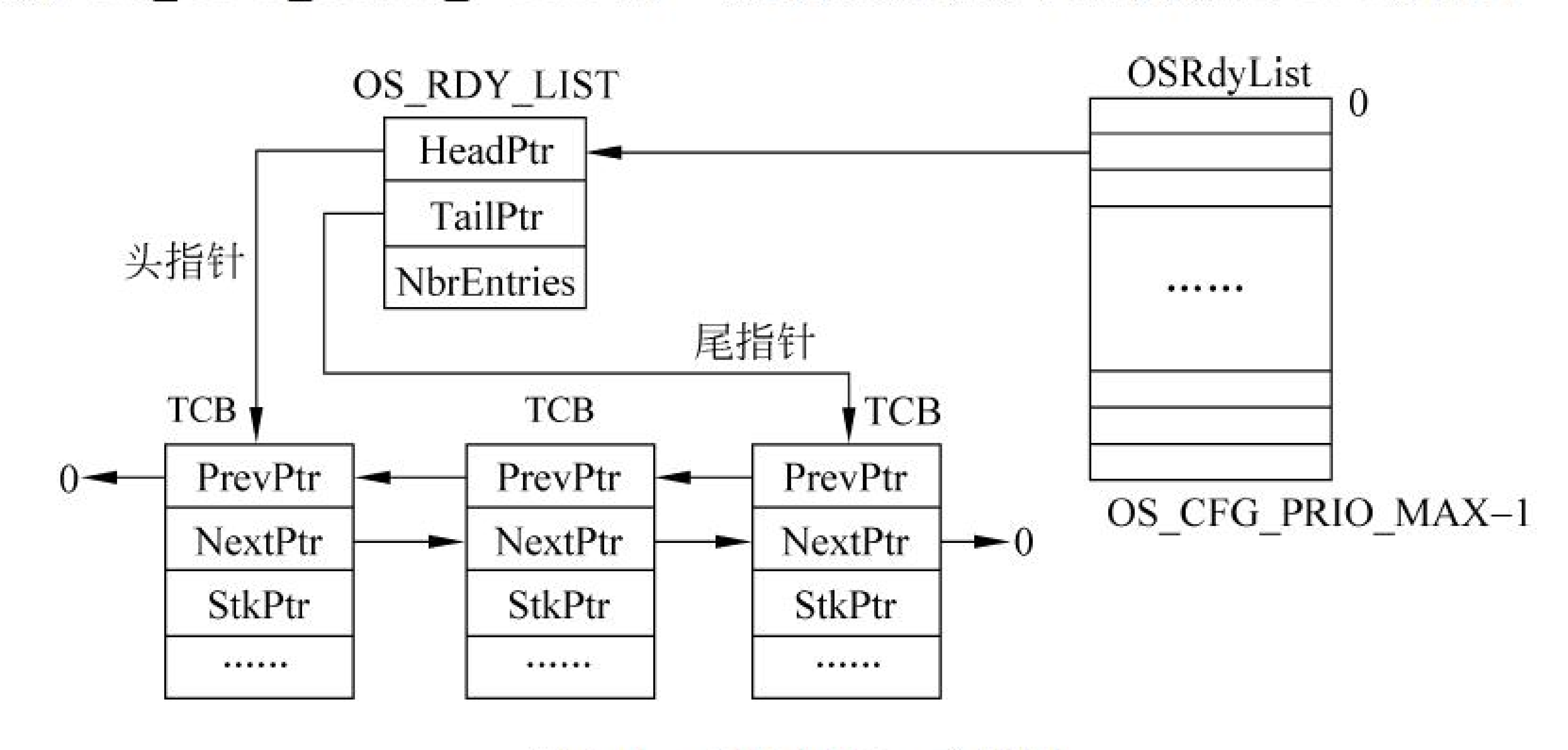


图 2.1 OSRdyList 示意图

2.2 μC/OS-Ⅲ内核任务管理分析

多任务管理其实就是在多个任务间调度和切换 CPU 使用权。在任务的执行过程中,由于 CPU 不断地切换,多任务管理系统看起来就像有多个 CPU 在执行工作,这样可以使得 CPU 的利用率最大化。

当 μC/OS-Ⅲ转向执行另一个任务时,它将当前任务的寄存器内容保存到堆栈中并从新任务的堆栈中将之前保存的数据复制到 CPU 寄存器中,这个过程叫做上下文切换。上下文切换需要一些开销。CPU 的寄存器越多,开销越大。上下文切换的时间基本取决于有多少个 CPU 寄存器需被存储和载入。在 μC/OS-Ⅲ中,任务切换时的堆栈设置类似于中断发生时的情况,即所有的 CPU 寄存器都被保存。假定任务堆栈中的信息将要被载入到 CPU 中,TSP 指向任务堆栈中最后一个被保存的寄存器。程序指针寄存器和状态寄存器最先被保存到任务堆栈中。ISP 指向当前中断堆栈的顶部。当中断服务程序被执行时,处理器把 R14 作为堆栈指针用于指向函数和局部参数。

任务可以创建其他任务,也可以挂起和恢复其他任务,向其他任务发送信号量和信息,与其他任务共享资源等。

从用户角度看,任务的状态共有五种:休眠态、就绪态、运行态、等待态、中断服务态。

休眠态:指任务已存在寄存器中,但不受系统的管理,可以通过任务创建函数接受系统的管理。当不需要这个任务时,可以删除任务,删除实际上是使该任务无法获取 CPU 的使用权。

就绪态:任务准备运行时,就进入就绪态,任务就绪表根据任务的优先级顺序对任务进行排序。

运行态:CPU 会调用当前就绪态中优先级最高的任务,使其获得 CPU 的使用权,但是

此时如果有更高优先级的任务就绪，CPU 会立即放弃该任务，调用更高优先级的任务，使其获得 CPU 的使用权。

等待待态：当用户调用使其进入等待某个事件的函数时，任务就会进入等待态，并自动放入等待表，待其等待的事件发生，就会自动进入就绪态，并放入就绪表。μC/OS-Ⅲ系统服务会判断这个就绪任务的优先级是否最高，如果是，CPU 会剥夺当前任务的 CPU 使用权，而刚就绪的任务会获得 CPU 的使用权。

中断服务态：CPU 允许中断，当中断发生，由于中断服务程序的优先级最高，所以 CPU 会保存当前正在运行的任务状态，然后进入中断服务程序。中断服务程序执行快一些，最好只是发送某个消息或信号。某个任务在消息队列中收到消息后，任务会进入就绪态，此时中断服务程序结束。CPU 查看任务就绪表中是否有更高优先级任务就绪，如果有，更高优先级的任务会进入到运行态，CPU 会进入到更高优先级的任务运行；如果就绪表中没有更高优先级的任务，CPU 会恢复到之前运行的任务状态，即恢复现场，回到之前运行的任务继续运行。各状态的转换如图 1.12 所示。

2.3　μC/OS-Ⅲ任务管理函数

2.3.1　任务创建 OSTaskCreate()，OSTaskCreateExt()

创建一个任务时必须为其分配一个 TCB、一个堆栈、一个优先级和其他一些参数。任务以无限循环的方式实现。另外，任务不允许有返回值。

在任务管理中一个重要的成员是 prio，即任务优先级，这个值越小，优先级越高。这个值的范围是不大于 OS_CFG_PRIO_MAX－2，且不小于 1。可以自己定义 OS_CFG_PRIO_MAX 宏定义大小。最高优先级和最低优先级分别被中断处理任务和空闲任务占据，虽然 μC/OS-Ⅲ允许多个任务共同占有同一个优先级，但是，如果任务占有最高和最低两个优先级，则会影响中断处理任务和空闲任务的性能。优先级最高任务只能在中断延迟功能关闭后使用，因为这时不存在中断处理任务。创建任务函数过程如图 2.2 所示。

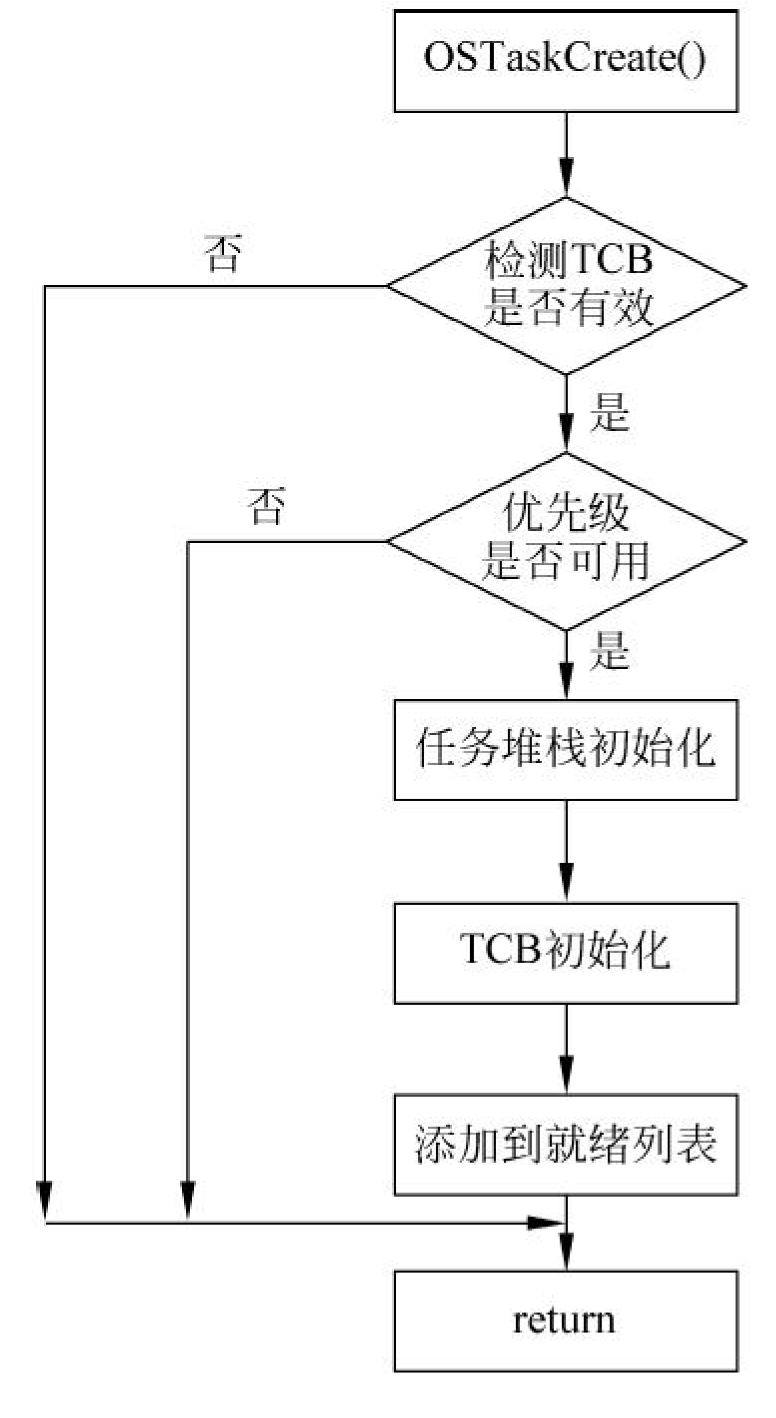
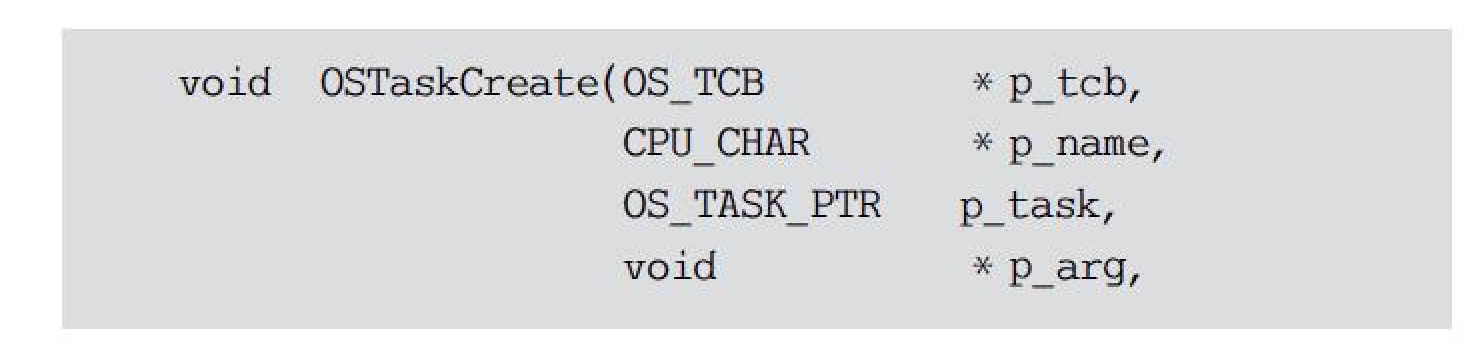


图 2.2　创建任务函数

OSTaskCreate()源代码如下：



```
void  OSTaskCreate(OS_TCB        * p_tcb,
                   CPU_CHAR      * p_name,
                   OS_TASK_PTR   p_task,
                   void          * p_arg,
```

```
                    OS_PRIO         prio,
                    CPU_STK         *p_stk_base,
                    CPU_STK_SIZE    stk_limit,
                    CPU_STK_SIZE    stk_size,
                    OS_MSG_QTY      q_size,
                    OS_TICK         time_quanta,
                    void            *p_ext,
                    OS_OPT          opt,
                    OS_ERR          *p_err)
{
    CPU_STK_SIZE   i;
#if OS_CFG_TASK_REG_TBL_SIZE > 0u
    OS_OBJ_QTY     reg_nbr;
#endif

    CPU_STK        *p_sp;
    CPU_STK        *p_stk_limit;
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508
    if (OSSafetyCriticalStartFlag == DEF_TRUE) {
        *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;
        return;
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {          //无法创建任务
        *p_err = OS_ERR_TASK_CREATE_ISR;
        return;
    }
#endif

#if OS_CFG_ARG_CHK_EN > 0u
    if (p_tcb == (OS_TCB *)0) {
        *p_err = OS_ERR_TCB_INVALID;
        return;
```

```
    }
    if (p_task == (OS_TASK_PTR)0) {
        *p_err = OS_ERR_TASK_INVALID;
        return;
    }
    if (p_stk_base == (CPU_STK *)0) {
        *p_err = OS_ERR_STK_INVALID;
        return;
    }
    if (stk_size < OSCfg_StkSizeMin) {                    //最小任务栈
        *p_err = OS_ERR_STK_SIZE_INVALID;
        return;
    }
    if (stk_limit >= stk_size) {                          //有效的栈约束
        *p_err = OS_ERR_STK_LIMIT_INVALID;
        return;
    }
    //优先级在 0 和 OS_CFG_PRIO_MAX - 1 之间
    if (prio >= OS_CFG_PRIO_MAX) {
        *p_err = OS_ERR_PRIO_INVALID;
        return;
    }
#endif

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    if (prio == (OS_PRIO)0) {
        if (p_tcb != &OSIntQTaskTCB) {
            *p_err = OS_ERR_PRIO_INVALID;                 //禁止使用 0 优先级
            return;
        }
    }
#endif

    if (prio == (OS_CFG_PRIO_MAX - 1u)) {
        if (p_tcb != &OSIdleTaskTCB) {
            *p_err = OS_ERR_PRIO_INVALID;                 //禁止使用空闲任务优先级
            return;
        }
    }

OS_TaskInitTCB(p_tcb);                                    //初始化 TCB

    *p_err = OS_ERR_NONE;                                 //清空任务栈
```

```
    if ((opt & OS_OPT_TASK_STK_CHK) != (OS_OPT)0) {
        if ((opt & OS_OPT_TASK_STK_CLR) != (OS_OPT)0) {
            p_sp = p_stk_base;
            for (i = 0u; i < stk_size; i++) {          //增长方向由高地址向低地址
                *p_sp = (CPU_STK)0;                    //自底向上清空栈
                p_sp++;
            }
        }
    }
                                                       //初始化栈结构
#if (CPU_CFG_STK_GROWTH == CPU_STK_GROWTH_HI_TO_LO)
    p_stk_limit = p_stk_base + stk_limit;
#else
    p_stk_limit = p_stk_base + (stk_size - 1u) - stk_limit;
#endif

    p_sp = OSTaskStkInit(p_task,
                         p_arg,
                         p_stk_base,
                         p_stk_limit,
                         stk_size,
                         opt);

    p_tcb->TaskEntryAddr = p_task;                     //存储任务入口地址
    p_tcb->TaskEntryArg  = p_arg;                      //存储入口参数
    p_tcb->NamePtr       = p_name;                     //存储任务名
    p_tcb->Prio          = prio;                       //存储任务优先级
    p_tcb->StkPtr        = p_sp;                       //存储新的栈顶指针
    p_tcb->StkLimitPtr   = p_stk_limit;
    p_tcb->TimeQuanta    = time_quanta;                //存储时间片数
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
    if (time_quanta == (OS_TICK)0) {
        p_tcb->TimeQuantaCtr = OSSchedRoundRobinDfltTimeQuanta;
    } else {
        p_tcb->TimeQuantaCtr = time_quanta;
    }
#endif
    p_tcb->ExtPtr        = p_ext;                      //TCB扩展指针
    p_tcb->StkBasePtr    = p_stk_base;                 //存储基地址
    p_tcb->StkSize       = stk_size;                   //存储栈大小
    p_tcb->Opt           = opt;

#if OS_CFG_TASK_REG_TBL_SIZE > 0u
    for (reg_nbr = 0u; reg_nbr < OS_CFG_TASK_REG_TBL_SIZE; reg_nbr++) {
```

```
            p_tcb->RegTbl[reg_nbr] = (OS_REG)0;
        }
    #endif

    #if OS_CFG_TASK_Q_EN > 0u
        OS_MsgQInit(&p_tcb->MsgQ, q_size);                    //初始化任务消息队列
    #endif

        OSTaskCreateHook(p_tcb);                              //添加任务到就绪表
        OS_CRITICAL_ENTER();
        OS_PrioInsert(p_tcb->Prio);
        OS_RdyListInsertTail(p_tcb);

    #if OS_CFG_DBG_EN > 0u
        OS_TaskDbgListAdd(p_tcb);
    #endif

        OSTaskQty++;

        if (OSRunning != OS_STATE_OS_RUNNING) {
            OS_CRITICAL_EXIT();
            return;
        }
        OS_CRITICAL_EXIT_NO_SCHED();
        OSSched();
    }
```

2.3.2 任务删除 OSTaskDel(),OSTaskDelReq()

任务的使命完成后,就要调用 OSTaskDel()删除该任务。OSTaskDel()实际上不是删除任务的代码,只是让任务不再具有使用 CPU 的资格。

OSTaskDel()源代码如下:

```
    #if OS_CFG_TASK_DEL_EN > 0u
    void  OSTaskDel(OS_TCB   *p_tcb,
                    OS_ERR   *p_err)
    {
        CPU_SR_ALLOC();

    #ifdef OS_SAFETY_CRITICAL
        if (p_err == (OS_ERR *)0) {
            OS_SAFETY_CRITICAL_EXCEPTION();
```

```
        return;
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        *p_err = OS_ERR_TASK_DEL_ISR;
        return;
    }
#endif

    if (p_tcb == &OSIdleTaskTCB) {                          //禁止删除空闲任务
        *p_err = OS_ERR_TASK_DEL_IDLE;
        return;
    }

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    if (p_tcb == &OSIntQTaskTCB) {                          //无法删除 ISR 处理任务
        *p_err = OS_ERR_TASK_DEL_INVALID;
        return;
    }
#endif

    if (p_tcb == (OS_TCB *)0) {
        CPU_CRITICAL_ENTER();
        p_tcb  = OSTCBCurPtr;
        CPU_CRITICAL_EXIT();
    }

    OS_CRITICAL_ENTER();
    switch (p_tcb->TaskState) {
        case OS_TASK_STATE_RDY:
             OS_RdyListRemove(p_tcb);
             break;

        case OS_TASK_STATE_SUSPENDED:
             break;

        case OS_TASK_STATE_DLY:                             //任务延迟
        case OS_TASK_STATE_DLY_SUSPENDED:
             OS_TickListRemove(p_tcb);
             break;

        case OS_TASK_STATE_PEND:
```

```
            case OS_TASK_STATE_PEND_SUSPENDED:
            case OS_TASK_STATE_PEND_TIMEOUT:
            case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
                 OS_TickListRemove(p_tcb);
                 switch (p_tcb->PendOn) {
                     case OS_TASK_PEND_ON_NOTHING:
                     case OS_TASK_PEND_ON_TASK_Q:
                     case OS_TASK_PEND_ON_TASK_SEM:
                          break;

                     case OS_TASK_PEND_ON_FLAG:          //移除等待列表
                     case OS_TASK_PEND_ON_MULTI:
                     case OS_TASK_PEND_ON_MUTEX:
                     case OS_TASK_PEND_ON_Q:
                     case OS_TASK_PEND_ON_SEM:
                          OS_PendListRemove(p_tcb);
                          break;

                     default:
                          break;
                 }
                 break;

            default:
                OS_CRITICAL_EXIT();
                 * p_err = OS_ERR_STATE_INVALID;
                return;
        }

#if OS_CFG_TASK_Q_EN > 0u
    (void)OS_MsgQFreeAll(&p_tcb->MsgQ);                  //释放任务消息队列
#endif

OSTaskDelHook(p_tcb);

#if OS_CFG_DBG_EN > 0u
    OS_TaskDbgListRemove(p_tcb);
#endif
    OSTaskQty--;
    OS_TaskInitTCB(p_tcb);                               //初始化 TCB
    p_tcb->TaskState = (OS_STATE)OS_TASK_STATE_DEL;
    OS_CRITICAL_EXIT_NO_SCHED();
    OSSched();                                           //查找新的最高优先级任务
     * p_err = OS_ERR_NONE;
}
#endif
```

删除任务函数过程如图 2.3 所示。

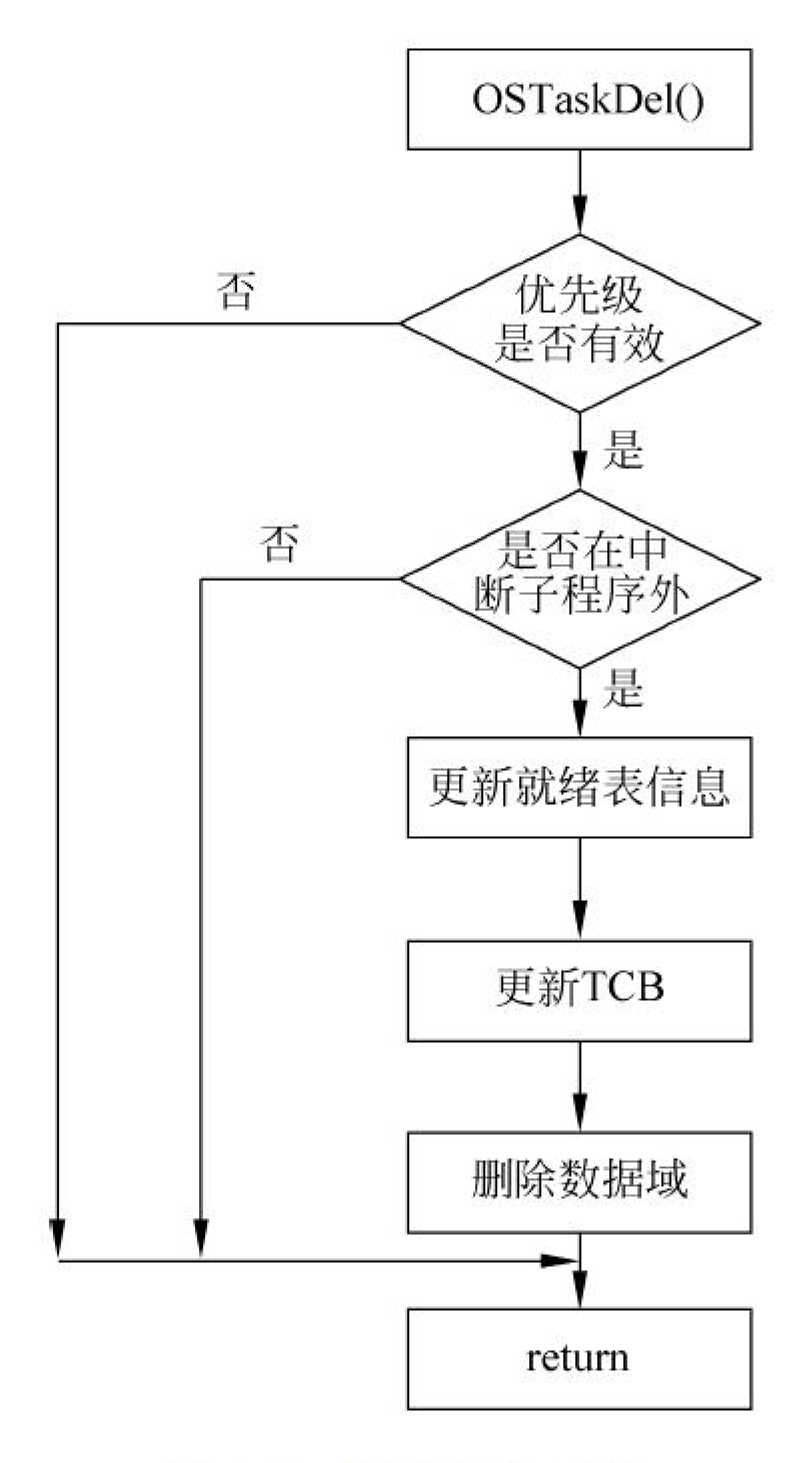


图 2.3 删除任务函数

2.3.3 任务挂起 OSTaskSuspend()

函数 OSTaskSuspend()用于挂起任务。挂起的含义相当于暂停,即剥夺任务的 CPU 使用权。可以多次调用 OSTaskSuspend()对任务进行挂起操作,即任务挂起是可以嵌套的。除空闲任务和延迟提交任务外,可以挂起其他任何任务。

OSTaskSuspend()源代码如下:

```
#if OS_CFG_TASK_SUSPEND_EN > 0u
void   OSTaskSuspend (OS_TCB   *p_tcb,
                      OS_ERR   *p_err)
{
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif
```

```
#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        *p_err = OS_ERR_TASK_SUSPEND_ISR;
        return;
    }
#endif

    if (p_tcb == &OSIdleTaskTCB) {                    //确定没有挂起空闲任务
        *p_err = OS_ERR_TASK_SUSPEND_IDLE;
        return;
    }

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    if (p_tcb == &OSIntQTaskTCB) {                    //不允许挂起 ISR 空闲任务
        *p_err = OS_ERR_TASK_SUSPEND_INT_HANDLER;
        return;
    }
#endif

    CPU_CRITICAL_ENTER();
    if (p_tcb == (OS_TCB *)0) {
        p_tcb = OSTCBCurPtr;
    }

    if (p_tcb == OSTCBCurPtr) {
        if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {  //调度器上锁
            CPU_CRITICAL_EXIT();
            *p_err = OS_ERR_SCHED_LOCKED;
            return;
        }
    }

    *p_err = OS_ERR_NONE;
    switch (p_tcb->TaskState) {
        case OS_TASK_STATE_RDY:
             OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
             p_tcb->TaskState   =   OS_TASK_STATE_SUSPENDED;
             p_tcb->SuspendCtr  =  (OS_NESTING_CTR)1;
             OS_RdyListRemove(p_tcb);
             OS_CRITICAL_EXIT_NO_SCHED();
             break;

        case OS_TASK_STATE_DLY:
```

```
                p_tcb->TaskState   = OS_TASK_STATE_DLY_SUSPENDED;
                p_tcb->SuspendCtr  = (OS_NESTING_CTR)1;
                CPU_CRITICAL_EXIT();
                break;

           case OS_TASK_STATE_PEND:
                p_tcb->TaskState  = OS_TASK_STATE_PEND_SUSPENDED;
                p_tcb->SuspendCtr  = (OS_NESTING_CTR)1;
                CPU_CRITICAL_EXIT();
                break;

           case OS_TASK_STATE_PEND_TIMEOUT:
                p_tcb->TaskState  = OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED;
                p_tcb->SuspendCtr  = (OS_NESTING_CTR)1;
                CPU_CRITICAL_EXIT();
                break;

           case OS_TASK_STATE_SUSPENDED:
           case OS_TASK_STATE_DLY_SUSPENDED:
           case OS_TASK_STATE_PEND_SUSPENDED:
           case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
                p_tcb->SuspendCtr++;
                CPU_CRITICAL_EXIT();
                break;

           default:
                CPU_CRITICAL_EXIT();
                *p_err  = OS_ERR_STATE_INVALID;
                return;
       }

       OSSched();
   }
   #endif
```

2.3.4 任务恢复 OSTaskResume()

函数 OSTaskResume()用来恢复挂起任务。调用该函数只需输入两个参数,一个是指向挂起任务的指针 p_tcb,一个是指向返回错误的指针 p_err。由于挂起是嵌套的,所以被恢复一次的任务不一定会解除挂起状态,需要等到嵌套层数为 0 时才会解除挂起状态。

OSTaskResume 源代码如下:

```
#if OS_CFG_TASK_SUSPEND_EN > 0u
void  OSTaskResume (OS_TCB   *p_tcb,
                    OS_ERR   *p_err)
{
    CPU_SR_ALLOC();
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
       *p_err = OS_ERR_TASK_RESUME_ISR;
        return;
    }
#endif

    CPU_CRITICAL_ENTER();
#if OS_CFG_ARG_CHK_EN > 0u
    if ((p_tcb == (OS_TCB *)0) ||
        (p_tcb == OSTCBCurPtr)) {
        CPU_CRITICAL_EXIT();
        *p_err = OS_ERR_TASK_RESUME_SELF;
        return;
    }
#endif

    *p_err  = OS_ERR_NONE;
    switch (p_tcb->TaskState) {
        case OS_TASK_STATE_RDY:
        case OS_TASK_STATE_DLY:
        case OS_TASK_STATE_PEND:
        case OS_TASK_STATE_PEND_TIMEOUT:
             CPU_CRITICAL_EXIT();
             *p_err = OS_ERR_TASK_NOT_SUSPENDED;
             break;

        case OS_TASK_STATE_SUSPENDED:
             OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
             p_tcb->SuspendCtr--;
             if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
                 p_tcb->TaskState = OS_TASK_STATE_RDY;
                 OS_TaskRdy(p_tcb);
             }
```

```
                OS_CRITICAL_EXIT_NO_SCHED();
                break;

           case OS_TASK_STATE_DLY_SUSPENDED:
                p_tcb->SuspendCtr--;
                if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
                    p_tcb->TaskState = OS_TASK_STATE_DLY;
                }
                CPU_CRITICAL_EXIT();
                break;

           case OS_TASK_STATE_PEND_SUSPENDED:
                p_tcb->SuspendCtr--;
                if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
                    p_tcb->TaskState = OS_TASK_STATE_PEND;
                }
                CPU_CRITICAL_EXIT();
                break;

           case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
                p_tcb->SuspendCtr--;
                if (p_tcb->SuspendCtr == (OS_NESTING_CTR)0) {
                    p_tcb->TaskState = OS_TASK_STATE_PEND_TIMEOUT;
                }
                CPU_CRITICAL_EXIT();
                break;

           default:
                CPU_CRITICAL_EXIT();
                *p_err = OS_ERR_STATE_INVALID;
                return;
       }
       OSSched();
   }
   #endif
```

2.4 μC/OS-Ⅲ任务管理应用开发

2.4.1 场景描述

假设某银行有 4 个窗口对外接待客户，从早上银行开门起不断有客户进入银行。由于每个窗口在某时刻只能接待一个客户，每个用户有存钱和取钱两种业务，在客户人数众多时需在任一窗口前顺次排队，对于刚进入银行的客户，如果某个窗口的业务员正空闲，则可上

前办理业务；反之，若 4 个窗口均有客户所占，他便会排在人数最少的队伍后面。现在需要编写一个程序来模拟银行的这种业务活动并计算一天中客户在银行的平均等待时间，并根据客户的存取信息结算当日金额。

2.4.2　设计总体架构和数据结构

总体架构设计和数据结构分析分别如图 2.4 和图 2.5 所示。

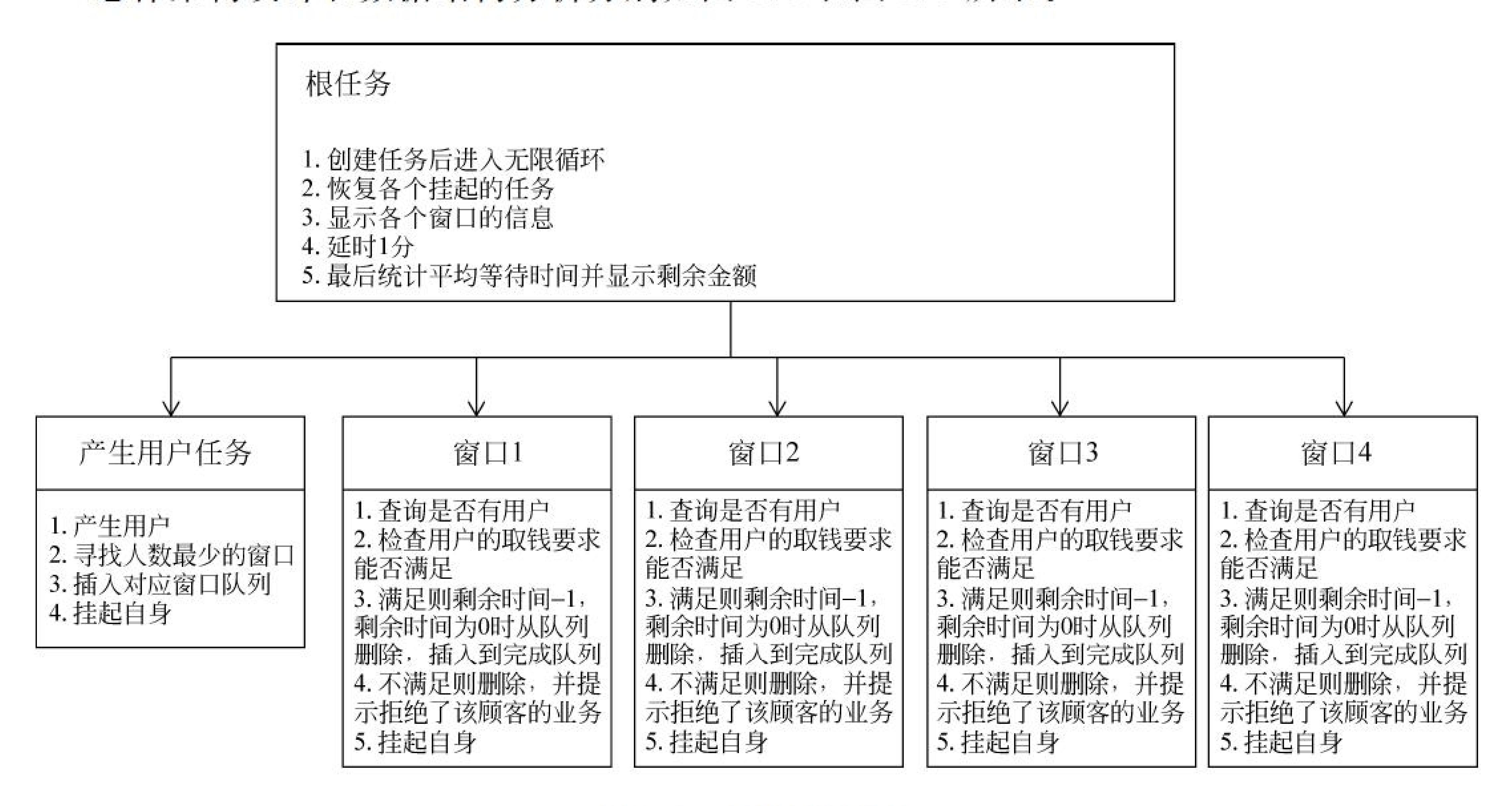


图 2.4　总体架构设计

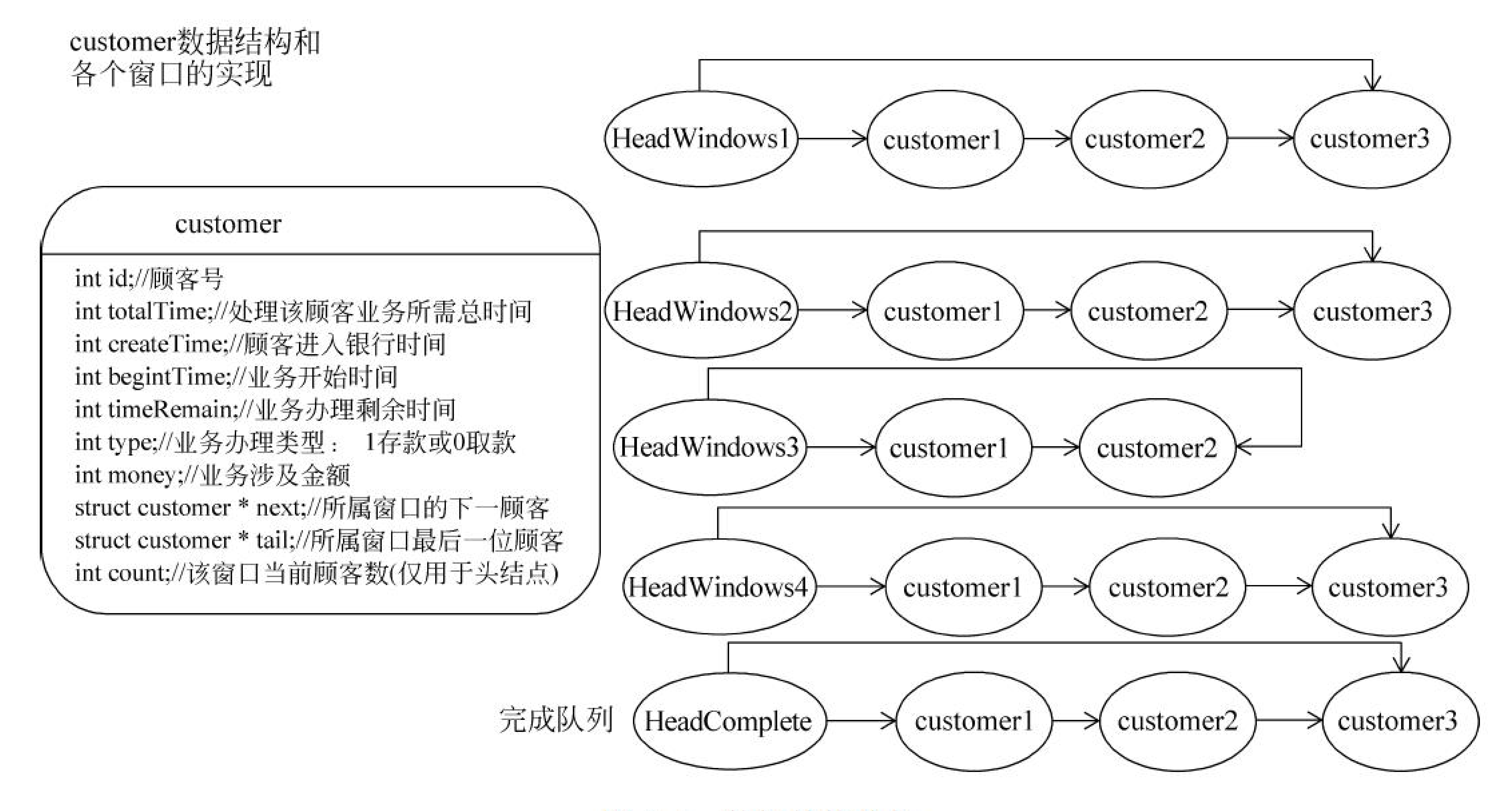


图 2.5　数据结构分析

2.4.3 代码实现

本实例采用的开发环境是 Windows 10＋VS2013，μC/OS 版本为 3.04.04。用到的机制有任务创建、任务挂起、任务恢复及任务延时等，并通过锁任务调度器的方法实现隔一定时间有序进行任务。

根任务代码如下所示：

```
while (1)
{
    OSSchedLock(&err);
    //使用 OSTaskResume()调用四个窗口任务
    OSTaskResume(&TaskWindow1TCB, &err);
    OSTaskResume(&TaskWindow2TCB, &err);
    OSTaskResume(&TaskWindow3TCB, &err);
    OSTaskResume(&TaskWindow4TCB, &err);
    //银行每天营业 10 个时间单位,假定每个时间单位进入一位顾客
    if (time <= 100)
        OSTaskResume(&TaskCustomerTCB, &err);
    time++;
    OSSchedUnlock(&err);
    APP_TRACE_DBG(("time: %d\n", time));    //打印当前时刻
    //打印窗口对应等待人数及窗口服务情况(空闲/正在办理业务/业务办理完毕)
    APP_TRACE_DBG(("窗口 1: "));
    PrintPeople(HeadWindow1.count);
    if (HeadWindow1.count == 0)
        APP_TRACE_DBG(("空闲\n"));
    else if ((*HeadWindow1.next).timeRemain == 0)
        APP_TRACE_DBG(("%d 号顾客业务办理完毕\n", (*HeadWindow1.next).id));
    else
        APP_TRACE_DBG(("%d 号顾客正在办理业务,还需 %d 分\n",
        (*HeadWindow1.next).id, (*HeadWindow1.next).timeRemain));
    ……
}
```

顾客产生函数如下所示：

```
//顾客任务,用于产生一位顾客
static void TaskCustomer(void * p_arg)
{
    OS_ERR  err;
```

```
    while (1)
    {
        OSSchedLock(&err);
        srand((unsigned)time);
        int timeRemain = rand() % 6 + 3;

//随机产生一位顾客办理业务所需时间,取值[3,8]
        int type = rand() % 2;   //为顾客随机分配一个业务类型
        //为顾客随机分配一个业务涉及金额,取值[1,10]
        int money = rand() % 10 + 1;
        //为每位顾客结构体分配一个内存空间
        struct customer * CreatedCustomer = (struct customer * )
                                    malloc(sizeof(struct customer));
        ( * CreatedCustomer).createTime = time;    //顾客进入银行时间
        ( * CreatedCustomer).id = totalId++;       //计算总的顾客人数
        ( * CreatedCustomer).money = money;
        ( * CreatedCustomer).type = type;
        ( * CreatedCustomer).timeRemain = timeRemain;
        ( * CreatedCustomer).totalTime = timeRemain;
        ( * CreatedCustomer).beginTime = -1;
        ( * CreatedCustomer).next = 0;
        int shortest = FindShortest();             //找到最短的等待队列并排在后面
        switch (shortest)
        {
        case 0:HeadWindow1.count++;                //该窗口排队人数 + 1
            ( * HeadWindow1.tail).next = CreatedCustomer;   //队尾的 next 指向此顾客
            HeadWindow1.tail = CreatedCustomer; break;      //队尾设为此顾客
    …
}
```

平均等待时间计算函数如下所示：

```
static void PrintWaitingTime()
{
    int i = 0;
    int total = 0;
    struct customer * tempPtr = HeadComplete.next;
    while (tempPtr != 0)
    {
        total += ( * tempPtr).beginTime - ( * tempPtr).createTime; //计算该顾客的等待时间
        tempPtr = ( * tempPtr).next;                               //移到下一位顾客
```

```
    }    float avg = (float)total / HeadComplete.count;    //取平均时间
    APP_TRACE_DBG((" % d 位顾客的平均等待时间是 % f 分钟\n",
                                                        HeadComplete.count,avg));
}
```

求最短队列函数如下所示：

```
//找到排队人数最少的窗口并返回该窗口号
static int FindShortest()
{
    OS_ERR  err;
    OSSchedLock(&err);
    int count[4];
    count[0] = HeadWindow1.count;
    count[1] = HeadWindow2.count;
    count[2] = HeadWindow3.count;
    count[3] = HeadWindow4.count;
    int shortest = 0;
    for (int i = 1; i < 4; i++)     //选择排序
    {
        if (count[i] < count[shortest])
        {
            shortest = i;
        }
    }
    OSSchedUnlock(&err);

    return shortest;
}
```

窗口处理函数如下所示：

```
//窗口任务,进行业务处理,具体表现为顾客业务处理剩余时间减 1
static void TaskWindow1(void * p_arg)
{
    OS_ERR  err;
    while (1){
        if (HeadWindow1.count > 0)                              //该窗口有顾客在办理业务
        {
            OSSchedLock(&err);                                  //锁调度
```

```
        int flag = 1;                                           //判断是否要处理
        int id = ( * HeadWindow1.next).id;                      //顾客 id
        int totaltime = ( * HeadWindow1.next).totalTime;        //顾客需要处理的总时间
        //初次处理
        if (( * HeadWindow1.next).totalTime == ( * HeadWindow1.next).timeRemain)
        {
            ( * HeadWindow1.next).beginTime = time;             //表示开始处理
            //判断金额变化
            if (( * HeadWindow1.next).type == 1){               //存钱
                APP_TRACE_DBG((" % d 号顾客存入 % d 万元\n", id,
                               ( * HeadWindow1.next).money));
                //该窗口钱数增加
                money_W1 = money_W1 + ( * HeadWindow1.next).money;
                money_Total += ( * HeadWindow1.next).money;     //总钱数增加
                flag = 1;                                       //可处理
            }
            else {
                //表示取钱数大于剩余钱数,无法处理该业务
                if (money_Total - ( * HeadWindow1.next).money < 0)
                {
                    APP_TRACE_DBG((" % d 号顾客欲取出 % d 万元被拒绝\n",
                                   id, ( * HeadWindow1.next).money));
//如果该顾客是窗口 1 的最后一名顾客
                    if (HeadWindow1.next == HeadWindow1.tail)
                    {
                        HeadWindow1.tail = &HeadWindow1;
                    }
//删除顾客
                    HeadWindow1.next = ( * HeadWindow1.next).next;
                    HeadWindow1.count -- ;
                    flag = 0;                                   //设为不处理
                }
                else
                {
                    APP_TRACE_DBG((" % d 号顾客取出 % d 万元\n", id,
                                   ( * HeadWindow1.next).money));
//该窗口钱数递减
                    money_W1 = money_W1 - ( * HeadWindow1.next).money;
//总钱数递减
                    money_Total -= ( * HeadWindow1.next).money;
                    flag = 1;
```

```
                    }
                }
            }
            if (flag == 1)                                  //可处理
            {
                if (( * HeadWindow1.next).timeRemain == 0)    //表示已经处理完毕
                {
                    //加入到处理完毕队列
                    ( * HeadComplete.tail).next = HeadWindow1.next;
                    HeadComplete.tail = HeadWindow1.next;
                    HeadComplete.count++;
//如果该顾客是窗口 1 的最后一名顾客
                    if (HeadWindow1.next == HeadWindow1.tail)
                    {
                        HeadWindow1.tail = &HeadWindow1;
                    }
                    HeadWindow1.next = ( * HeadWindow1.next).next;    //删除顾客
                    HeadWindow1.count--;
                }
                else
                {
                    ( * HeadWindow1.next).timeRemain--;       //处理剩余时间减少
                }
            }
            OSSchedUnlock(&err);                          //调度器解除锁定
        }
        OSTaskSuspend(&TaskWindow1TCB, &err);             //挂起自己
    }

}
```

习题

1. μC/OS-Ⅲ的任务控制块 OS_TCB 主要由哪些元素组成？
2. 任务控制块被哪一个内核结构统一管理？
3. μC/OS-Ⅲ可以定义多少个任务？
4. μC/OS-Ⅲ有多少个任务状态？各个任务状态之间在何时发生切换？
5. 任务创建函数 OSTaskCreate()和 OSTaskCreateExt()的区别是什么？
6. 在任务创建函数中，内核创建任务的主要工作是什么？

7. 任务删除函数 OSTaskDel()和 OSTaskDelReq()的区别是什么?

8. 在任务删除函数中,内核删除任务的主要工作是什么?

9. 调用任务挂起函数 OSTaskSuspend()后,调用哪个函数能够使任务恢复运行?

10. 在任务挂起函数 OSTaskSuspend()中,内核的主要工作是什么?

11. 在任务恢复函数 OSTaskResume()中,内核的主要工作是什么?

第 3 章



μC/OS-Ⅲ内核调度

3.1 μC/OS-Ⅲ内核调度机制

μC/OS-Ⅲ内核调度主要是协调任务之间对 CPU 的使用权。具体而言，μC/OS-Ⅲ任务调度分为抢占式调度和时间片轮转调度。其中抢占式调度根据发生的时间又分为任务级任务调度 OSSched()和中断级任务调度 OSIntExit()，由于调度的时机不同，这两种任务调度对应的上下文切换也有所不同，后面的章节会详细介绍。

时间片轮转调度，本身可以通过编译开关的配置选择启用或者不启用。启用状态下，当用于提供系统时钟的定时器发生中断(一个时钟节拍过去)时，该中断服务函数会调用 OSTimeTick()函数，在该函数内部系统会更新时基的相关数据结构，同时调用时间片轮转调度函数 OS_SchedRoundRobin()。

在优先级调度的过程中离不开数据结构：任务就绪表 OSPrioTbl。μC/OS-Ⅲ对优先级的数量无限制，由宏定义 OS_CFG_PRIO_MAX 确定。然而，配置 μC/OS-Ⅲ的优先级在 32～256 之间已经能满足大多数的应用。OSPrioTbl 结构如图 3.1 所示，最高优先级 0 处于 OSPrioTbl[0]中，最低优先级 OS_CFG_PRIO_MAX－1 处于 OSPrioTbl[OS_PROI_SIZE－1]中。

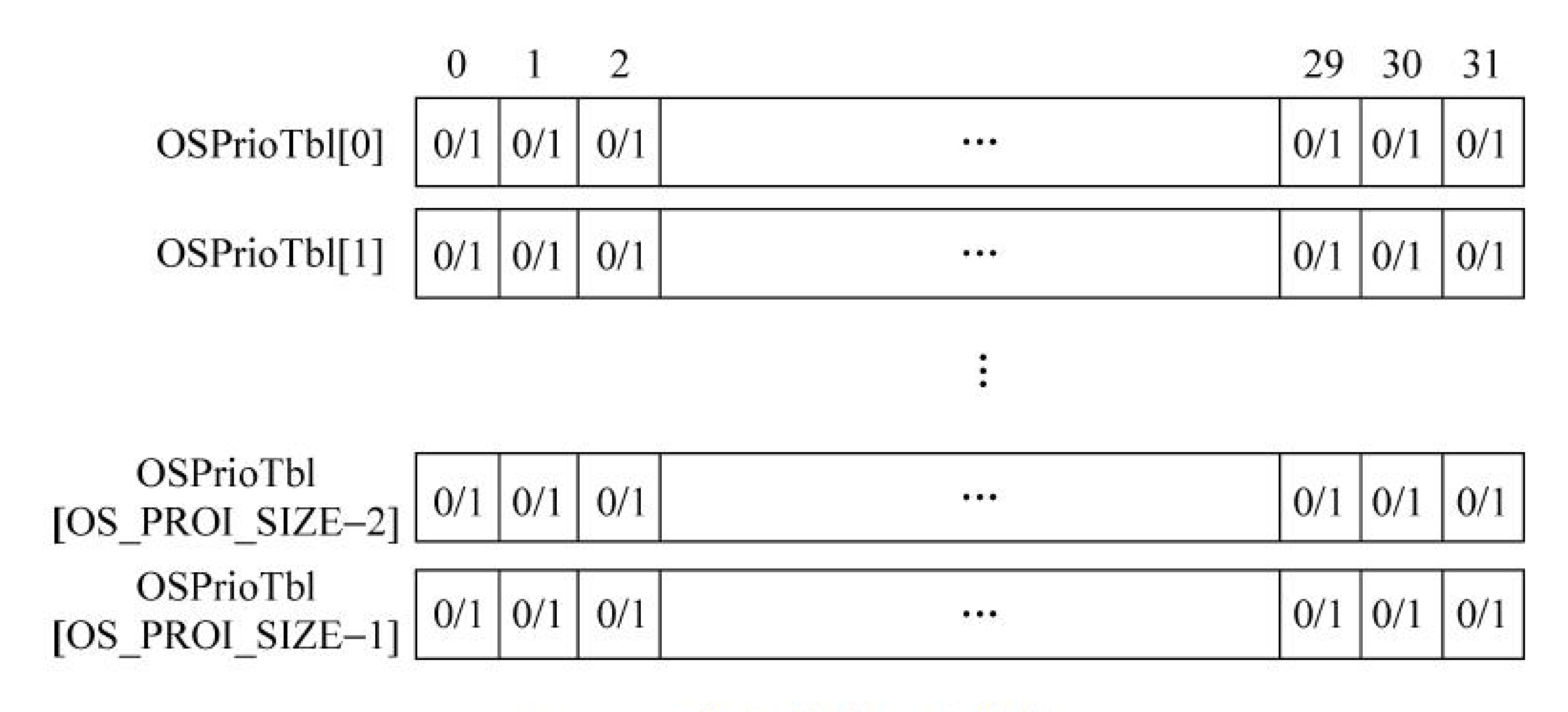


图 3.1　任务优先级表结构

3.2 μC/OS-Ⅲ内核抢占优先级调度分析

μC/OS-Ⅲ中比较常用的两个抢占式任务调度函数是OSSched()和OSIntExit(),这两个函数的抢占时机分别是正常运行状态和退出中断时。二者的函数主体比较相似,都是先判断是否满足调度条件,之后通过调用用函数OS_PrioGetHighest()来获取当前就绪队列中最高优先级任务所持有的优先级,然后根据任务TCB进行上下文切换。

一个简单的抢占过程如图3.2所示。

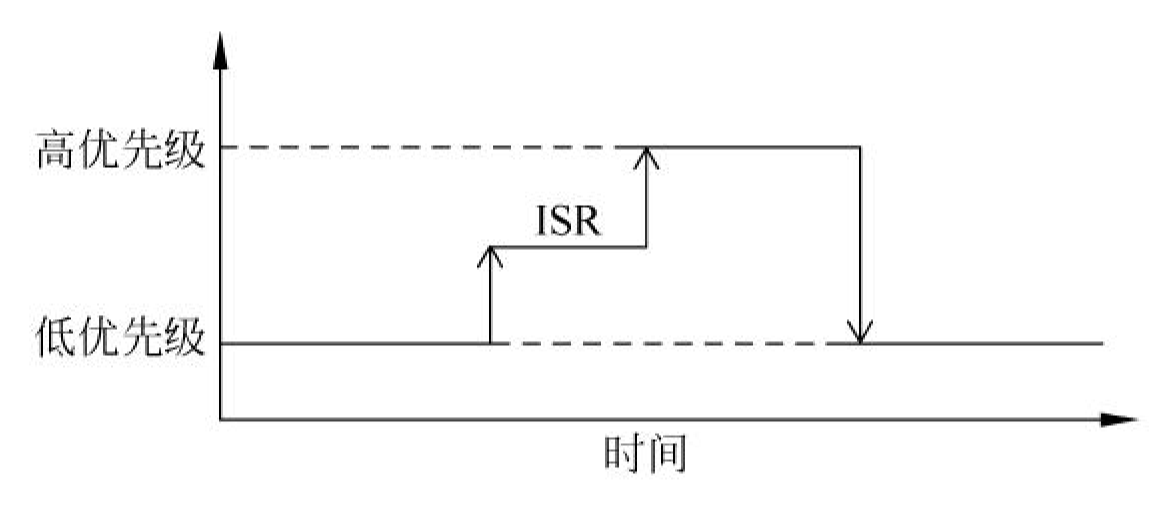


图3.2 抢占过程

其中,获取最高优先级函数OS_PrioGetHighest()源码如下:

```
OS_PRIO  OS_PrioGetHighest (void)
{
    CPU_DATA   * p_tbl;
    OS_PRIO     prio;

    prio   = (OS_PRIO)0;
    p_tbl  = &OSPrioTbl[0];
    //通过位映像表查找最高优先级
    while ( * p_tbl == (CPU_DATA)0) {
        //计算 CPU_DATA 入口
        prio += DEF_INT_CPU_NBR_BITS;
        p_tbl++;
    }
    //找到入口处第一个位组
    prio += (OS_PRIO)CPU_CntLeadZeros( * p_tbl);
    return (prio);
}
```

当最高优先级的任务被确定后,执行上下文切换,OSSched()调用的是宏OS_TASK_SW(),这里实际上调用的是汇编函数OSCtexSw(),而OSIntExit()调用的是汇编函数OSIntCtxSw()。

OSSched()源代码如下:

```
void  OSSched(void)
{
    CPU_SR_ALLOC();

    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        return;
    }
    //调度器上锁
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
        return;
    }

    CPU_INT_DIS();
    //就绪最高优先级任务
    OSPrioHighRdy    = OS_PrioGetHighest();
    OSTCBHighRdyPtr  = OSRdyList[OSPrioHighRdy].HeadPtr;
    //当前任务优先级是否最高
    if (OSTCBHighRdyPtr == OSTCBCurPtr) {
        CPU_INT_EN();
        return;
    }

#if OS_CFG_TASK_PROFILE_EN > 0u
    OSTCBHighRdyPtr->CtxSwCtr++;          //上下文切换计数
#endif
    OSTaskCtxSwCtr++;
    OS_TASK_SW();                         //执行任务级上下文切换
    CPU_INT_EN();
}
```

OSIntExit()源代码如下：

```
void  OSIntExit(void)
{
    CPU_SR_ALLOC();

    if (OSRunning != OS_STATE_OS_RUNNING) {
        return;
    }

    CPU_INT_DIS();
    if (OSIntNestingCtr == (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }
```

```
    OSIntNestingCtr--;
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }

    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }

    OSPrioHighRdy    = OS_PrioGetHighest();
    OSTCBHighRdyPtr  = OSRdyList[OSPrioHighRdy].HeadPtr;
    if (OSTCBHighRdyPtr == OSTCBCurPtr) {
        CPU_INT_EN();
        return;
    }

#if OS_CFG_TASK_PROFILE_EN > 0u
    OSTCBHighRdyPtr->CtxSwCtr++;
#endif
    OSTaskCtxSwCtr++;                           //记录交换次数

    OSIntCtxSw();                               //执行中断级切换
    CPU_INT_EN();
}
```

3.3　μC/OS-Ⅲ内核时间片轮转调度分析

μC/OS-Ⅲ引入了时间片轮转调度。当任务列表中处于最高优先级的任务不止一个时，内核对这几个任务进行时间片轮转调度。在运行之前会给任务分配时间片，时间片越多，连续占用 CPU 的时间越长。分配的过程需要调用 OSSchedRoundRobinCfg()函数，该函数先进行时间片轮转调度的初始化。一个时间片轮转的例子如图 3.3 所示。

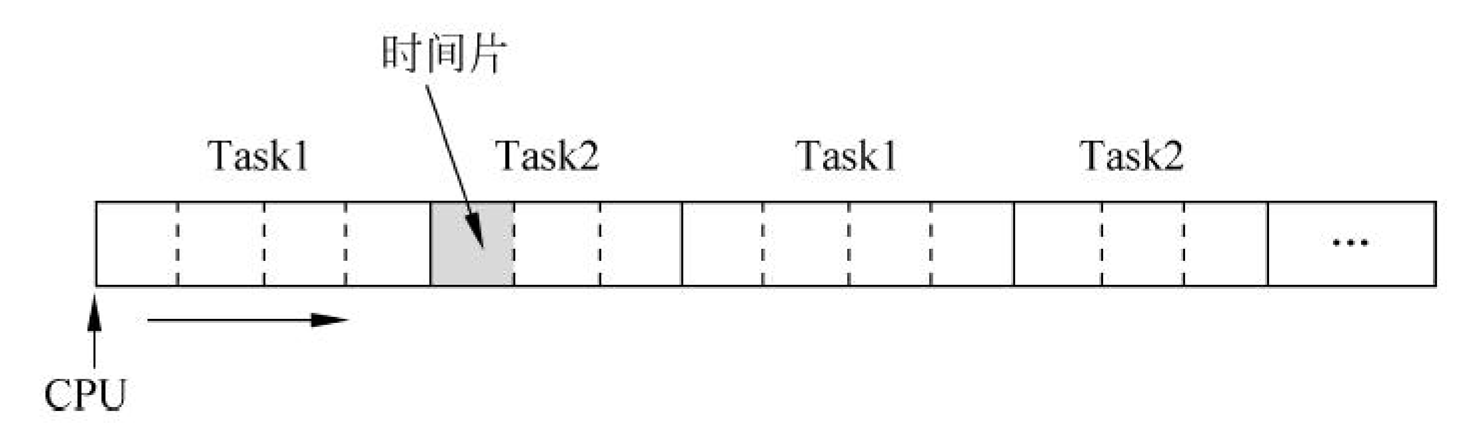


图 3.3　时间片轮转示意图

OSSchedRoundRobinCfg()的源代码如下：

```
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
void  OSSchedRoundRobinCfg(CPU_BOOLEAN   en,
                           OS_TICK       dflt_time_quanta,
                           OS_ERR        *p_err)
{
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

    CPU_CRITICAL_ENTER();
    if (en != DEF_ENABLED) {
        OSSchedRoundRobinEn = DEF_DISABLED;
    } else {
        OSSchedRoundRobinEn = DEF_ENABLED;
    }

    if (dflt_time_quanta > (OS_TICK)0) {
        OSSchedRoundRobinDfltTimeQuanta = dflt_time_quanta;
    } else {
        OSSchedRoundRobinDfltTimeQuanta = (OS_TICK)(OSCfg_TickRate_Hz / (OS_RATE_HZ)10);
    }
    CPU_CRITICAL_EXIT();
   *p_err = OS_ERR_NONE;
}
#endif
```

时间片轮转调度初始化结束，开始任务调度。每过一个时间单位，OS_SchedRoundRobin()函数都会被调用。在该函数内，当前任务的时间片会减1，并在函数内检测当前任务所持有的时间片是否已用完，如果用完则把当前任务的TCB从就绪表的表头移至表尾。但是OS_SchedRoundRobin()并不直接进行任务调度，因为此时处于定时器中断内，待退出中断后，中断级任务调度函数会调用并查找任务链表中优先级最高的任务，选择最高优先级任务进入运行态，实现调度。

OS_SchedRoundRobin()源代码如下：

```
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
void  OS_SchedRoundRobin(OS_RDY_LIST   *p_rdy_list)
{
```

```
OS_TCB    *p_tcb;
CPU_SR_ALLOC();

if (OSSchedRoundRobinEn != DEF_TRUE) {        //确定轮转使能
    return;
}

CPU_CRITICAL_ENTER();
p_tcb = p_rdy_list->HeadPtr;                  //减少时间份额计数器

if (p_tcb == (OS_TCB *)0) {
    CPU_CRITICAL_EXIT();
    return;
}

if (p_tcb == &OSIdleTaskTCB) {
    CPU_CRITICAL_EXIT();
    return;
}

if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {
    p_tcb->TimeQuantaCtr--;
}

if (p_tcb->TimeQuantaCtr > (OS_TICK)0) {      //任务未完成
    CPU_CRITICAL_EXIT();
    return;
}

if (p_rdy_list->NbrEntries < (OS_OBJ_QTY)2) {
    CPU_CRITICAL_EXIT();                      //仅限同优先级任务
    return;
}
//调度器上锁
if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
    CPU_CRITICAL_EXIT();
    return;
}
//将当前 OS_TCB 移至表尾
OS_RdyListMoveHeadToTail(p_rdy_list);
//指向新的表头 OS_TCB
p_tcb = p_rdy_list->HeadPtr;
if (p_tcb->TimeQuanta == (OS_TICK)0) {        //默认时间片
    p_tcb->TimeQuantaCtr = OSSchedRoundRobinDfltTimeQuanta;
} else {
    //存入新的时间片份额
```

```
            p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;
        }
        CPU_CRITICAL_EXIT();
    }
    #endif
```

当任务的工作完成时，如果时间片还有剩余，任务可以放弃剩余的时间片，交出 CPU 的使用权，交给同优先级的其他任务使用。此时需要调用放弃时间片函数 OSSchedRoundRobinYield()，但需要注意该函数不会将 CPU 的使用权直接交给低优先级的任务。

OSSchedRoundRobinYield()源代码如下：

```
    #if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
    void  OSSchedRoundRobinYield (OS_ERR   *p_err)
    {
        OS_RDY_LIST   *p_rdy_list;
        OS_TCB        *p_tcb;
        CPU_SR_ALLOC();

    #ifdef OS_SAFETY_CRITICAL
        if (p_err == (OS_ERR *)0) {
            OS_SAFETY_CRITICAL_EXCEPTION();
            return;
        }
    #endif

    #if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
        if (OSIntNestingCtr > (OS_NESTING_CTR)0) {         //ISR 中不能调用
           *p_err = OS_ERR_YIELD_ISR;
            return;
        }
    #endif

        if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
           *p_err = OS_ERR_SCHED_LOCKED;
            return;
        }

        if (OSSchedRoundRobinEn != DEF_TRUE) {
           *p_err = OS_ERR_ROUND_ROBIN_DISABLED;
            return;
        }

        CPU_CRITICAL_ENTER();
```

```
        p_rdy_list = &OSRdyList[OSPrioCur];     //一个任务无法放弃时间片
        if (p_rdy_list->NbrEntries < (OS_OBJ_QTY)2) {
            CPU_CRITICAL_EXIT();
           *p_err = OS_ERR_ROUND_ROBIN_1;
            return;
        }

        OS_RdyListMoveHeadToTail(p_rdy_list);
        p_tcb = p_rdy_list->HeadPtr;
        if (p_tcb->TimeQuanta == (OS_TICK)0) {
            p_tcb->TimeQuantaCtr = OSSchedRoundRobinDfltTimeQuanta;
    }
    else {
            p_tcb->TimeQuantaCtr = p_tcb->TimeQuanta;
        }

        CPU_CRITICAL_EXIT();
        OSSched();
       *p_err = OS_ERR_NONE;
    }
    #endif
```

3.4　μC/OS-Ⅲ内核调度管理函数

1. OSInit()：系统内核初始化函数

```
    void OSInit(OS_ERR  *p_err)
    {
        CPU_STK       *p_stk;
        CPU_STK_SIZE  size;

    #ifdef OS_SAFETY_CRITICAL
        if (p_err == (OS_ERR *)0) {
            OS_SAFETY_CRITICAL_EXCEPTION();
            return;
        }
    #endif

        OSInitHook();

        OSIntNestingCtr = (OS_NESTING_CTR)0;            //清空中断嵌套数

        OSRunning  =  OS_STATE_OS_STOPPED;              //未开始多任务调度
```

```
    //清空调度锁计数
    OSSchedLockNestingCtr            = (OS_NESTING_CTR)0;

#if OS_CFG_SCHED_LOCK_TIME_MEAS_EN > 0u
    OSSchedLockTimeBegin             = (CPU_TS)0;
    OSSchedLockTimeMax               = (CPU_TS)0;
    OSSchedLockTimeMaxCur            = (CPU_TS)0;
#endif

#ifdef OS_SAFETY_CRITICAL_IEC61508
    OSSafetyCriticalStartFlag        =  DEF_FALSE;
#endif

#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
    OSSchedRoundRobinEn              = DEF_FALSE;
    OSSchedRoundRobinDfltTimeQuanta = OSCfg_TickRate_Hz / 10u;
#endif

    if (OSCfg_ISRStkSize > (CPU_STK_SIZE)0) {
        p_stk = OSCfg_ISRStkBasePtr;                  //清空异常栈
        if (p_stk != (CPU_STK *)0) {
            size  = OSCfg_ISRStkSize;
            while (size > (CPU_STK_SIZE)0) {
                size--;
                *p_stk = (CPU_STK)0;
                p_stk++;
            }
        }
    }

#if OS_CFG_APP_HOOKS_EN > 0u
    //清空应用程序指针
    OS_AppTaskCreateHookPtr = (OS_APP_HOOK_TCB )0;
    OS_AppTaskDelHookPtr     = (OS_APP_HOOK_TCB )0;
    OS_AppTaskReturnHookPtr = (OS_APP_HOOK_TCB )0;

    OS_AppIdleTaskHookPtr    = (OS_APP_HOOK_VOID)0;
    OS_AppStatTaskHookPtr    = (OS_APP_HOOK_VOID)0;
    OS_AppTaskSwHookPtr      = (OS_APP_HOOK_VOID)0;
    OS_AppTimeTickHookPtr    = (OS_APP_HOOK_VOID)0;
#endif

    OS_PrioInit();                          //初始化优先级位映像表

    OS_RdyListInit();                       //初始化就绪列表

    OS_TaskInit(p_err);                     //初始化任务管理器
```

```
    if ( * p_err != OS_ERR_NONE) {
        return;
    }

#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    OS_IntQTaskInit(p_err);                    //初始化中断队列控制任务
    if ( * p_err != OS_ERR_NONE) {
        return;
    }
#endif

    OS_IdleTaskInit(p_err);                    //初始化空闲任务
    if ( * p_err != OS_ERR_NONE) {
        return;
    }

    OS_TickTaskInit(p_err);                    //初始化时钟节拍任务
    if ( * p_err != OS_ERR_NONE) {
        return;
    }

#if OS_CFG_STAT_TASK_EN > 0u
    OS_StatTaskInit(p_err);
    if ( * p_err != OS_ERR_NONE) {
        return;
    }
#endif

#if OS_CFG_FLAG_EN > 0u                        //初始化事件标志模块
    OS_FlagInit(p_err);
    if ( * p_err != OS_ERR_NONE) {
        return;
    }
#endif


#if OS_CFG_MEM_EN > 0u                         //初始化内存管理模块
    OS_MemInit(p_err);
    if ( * p_err != OS_ERR_NONE) {
        return;
    }
#endif

#if (OS_MSG_EN) > 0u                           //初始化 OS_MSG 空闲列表
    OS_MsgPoolInit(p_err);
    if ( * p_err != OS_ERR_NONE) {
```

```
            return;
        }
    #endif

    #if OS_CFG_MUTEX_EN > 0u                    //初始化互斥量管理模块
        OS_MutexInit(p_err);
        if ( * p_err != OS_ERR_NONE) {
            return;
        }
    #endif


    #if OS_CFG_Q_EN > 0u
        OS_QInit(p_err);                        //初始化消息队列模块
        if ( * p_err != OS_ERR_NONE) {
            return;
        }
    #endif


    #if OS_CFG_SEM_EN > 0u                      //初始化信号量管理模块
        OS_SemInit(p_err);
        if ( * p_err != OS_ERR_NONE) {
            return;
        }
    #endif

    #if OS_CFG_TMR_EN > 0u                      //初始化定时器管理模块
        OS_TmrInit(p_err);
        if ( * p_err != OS_ERR_NONE) {
            return;
        }
    #endif

    #if OS_CFG_DBG_EN > 0u
        OS_Dbg_Init();
    #endif

        OSCfg_Init();
    }
```

2．OSSchedLock()：对任务调度器上锁，禁止任务级调度

```
void  OSSchedLock(OS_ERR   * p_err)
{
```

```
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        *p_err = OS_ERR_SCHED_LOCK_ISR;
        return;
    }
#endif

    if (OSRunning != OS_STATE_OS_RUNNING) {
        *p_err = OS_ERR_OS_NOT_RUNNING;
        return;
    }
    /* 防止 OSSchedLockNestingCtr 溢出 */
    if (OSSchedLockNestingCtr >= (OS_NESTING_CTR)250u) {
        *p_err = OS_ERR_LOCK_NESTING_OVF;
        return;
    }

    CPU_CRITICAL_ENTER();
    OSSchedLockNestingCtr++;                    //增加嵌套锁层数
#if OS_CFG_SCHED_LOCK_TIME_MEAS_EN > 0u
    OS_SchedLockTimeMeasStart();
#endif
    CPU_CRITICAL_EXIT();
    *p_err = OS_ERR_NONE;
}
```

3．OSSchedUnlock()：对任务调度器解锁，总调用次数与上锁次数相同

```
void  OSSchedUnlock(OS_ERR  *p_err)
{
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
```

```
    }
#endif

#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        *p_err = OS_ERR_SCHED_UNLOCK_ISR;
        return;
    }
#endif

    if (OSRunning != OS_STATE_OS_RUNNING) {
        *p_err = OS_ERR_OS_NOT_RUNNING;
        return;
    }

    if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {
        *p_err = OS_ERR_SCHED_NOT_LOCKED;
        return;
    }

    CPU_CRITICAL_ENTER();
    OSSchedLockNestingCtr--;                //减少嵌套锁层数
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
        CPU_CRITICAL_EXIT();
        *p_err = OS_ERR_SCHED_LOCKED;
        return;
    }

#if OS_CFG_SCHED_LOCK_TIME_MEAS_EN > 0u
    OS_SchedLockTimeMeasStop();
#endif

    CPU_CRITICAL_EXIT();                    //再次启动调度器
    OSSched();                              //执行调度
   *p_err = OS_ERR_NONE;
}
```

4. OSStart()：开始多任务调度

```
void  OSStart(OS_ERR  *p_err)
{
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
```

```
    }
#endif

    if (OSRunning == OS_STATE_OS_STOPPED) {
        //查找最高优先级任务
        OSPrioHighRdy    = OS_PrioGetHighest();
        OSPrioCur        = OSPrioHighRdy;
        OSTCBHighRdyPtr  = OSRdyList[OSPrioHighRdy].HeadPtr;
        OSTCBCurPtr      = OSTCBHighRdyPtr;
        OSRunning        = OS_STATE_OS_RUNNING;
        OSStartHighRdy();                    //运行具有最高优先级的就绪任务
       *p_err            = OS_ERR_FATAL_RETURN;
    } else {
       *p_err            = OS_ERR_OS_RUNNING;
    }
}
```

习题

1. μC/OS-Ⅲ任务调度和中断调度的区别是什么?
2. 任务优先级表 OSPrioTbl 的结构是什么?
3. 获取就绪队列中最高任务优先级的函数是什么?该函数在内核中的主要作用是什么?
4. 时间片轮转调度法是什么?μC/OS-Ⅲ内核是如何实现这一调度机制的?
5. μC/OS-Ⅲ时间片轮转调度由哪几部分组成?各部分的功能是什么?
6. 内核初始化函数 OSInit()在初始化过程中做了哪些主要工作?
7. 调度器加锁函数 OSSchedLock()是如何给调度器上锁的?
8. 调度器解锁函数 OSSchedUnLock()是如何给调度器解锁的?
9. OSStart()是如何开始任务调度的?

第 4 章



μC/OS-Ⅲ任务间同步机制

4.1 μC/OS-Ⅲ任务同步机制

为了实现各个任务之间合作和无冲突运行,在各任务之间必须建立一些约束关系。

一是各任务间应该具有一种互斥关系,即对于某个共享资源,如果一个任务正在使用,则其他任务只能等待,等到该任务释放该资源后,等待的任务之一才能使用它。

二是相关的任务在执行时要有先后次序,一个任务要等其伙伴发来通知,或建立了某个条件后才能继续执行,否则只能等待。

任务之间的这种制约的合作运行机制叫做任务间的同步。

在多任务系统中,任务之间既存在着相互制约也存在着相互合作。每个任务在执行时都或多或少地要使用一些系统的公共资源,但系统的公共资源是有限的,如果多个任务并发去执行,系统不可能同时满足所有任务的资源要求,所以并发执行的任务间需要同步机制来进行协调,使多任务无冲突地执行。一个完善的多任务操作系统需要具备良好的同步通信机制来协调多任务对共享资源进行有序地访问,保证任务都能访问资源并且不影响任务的独立性。同时多任务系统可以实现多个任务共同合作来完成一个功能需求,这就要求任务间执行要有一定的先后次序,只有当前一个任务执行完成后才会有下一个任务接收到通知继续执行,这也需要任务间的同步机制。

μC/OS-Ⅲ中应用到的与同步机制相关的结构包括信号量、互斥体、事件标志组和消息队列。

4.2 μC/OS-Ⅲ信号量机制分析

信号量的用途主要有三个:

第一,表示一个或多个事件的发生;

第二,表示公共资源的可用状态(二值信号量,类似互斥量,但互斥量主要不是用来解决

此问题)；

第三，表示一个或多个相同资源的访问(多值信号量)以及可用的数量。

对信号量的操作都是原子性的，所以可以通过改变并识别信号量的状态来决定资源是否可用，从而实现对资源原子性的操作。信号量机制如同一种上锁机制，提供对临界资源的控制，其中所谓的临界资源指的是在同一时刻只能被一个任务所使用的资源。

信号量用于对临界资源保护的同时也可以用来进行任务之间的同步。简单信号量机制如图 4.1 所示。

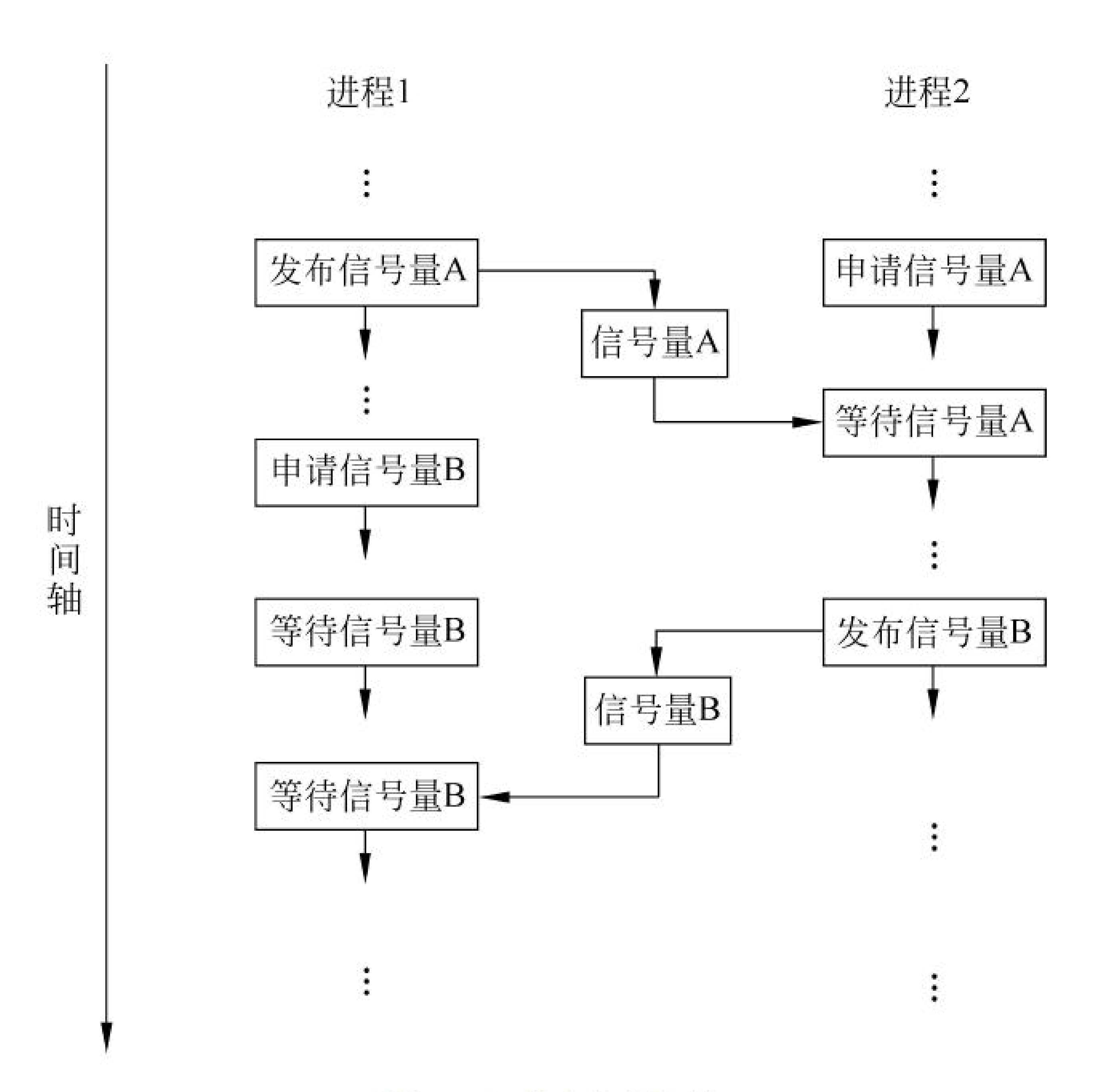


图 4.1 信号量机制

信号量可以看作是一个整型变量，信号量的初始值为临界区公共资源的个数。每当一个任务申请信号量时，其实是在申请使用一份资源(P 操作)，如果信号量的值大于 0，则信号量的值减 1，系统为该任务分配资源并执行此任务；如果信号量的值小于等于 0，则信号量的值减 1，任务挂起，进入等待队列。当有任务使用完资源时，任务释放信号量(V 操作)，从任务等待队列中唤醒一个任务，对其分配资源，执行新任务；如果任务等待队列中没有任务，则信号量的值加 1。对信号量的操作是原子性的，即不可分割的，所以同一时刻只有一个任务对信号量进行操作。当信号量的值小于 0 时，信号量的绝对值则表示正在任务队列等待的任务个数。

μC/OS 对信号量进行了一些封装，添加了一些其他数据来完善信号量，并定义了一些函数来对信号量进行操作。

4.2.1 μC/OS-Ⅲ信号量数据结构

(1) 在代码中经常可见用 OS_SEM 定义信号量。

```
typedef struct os_sem OS_SEM;
```

μC/OS 将信号量结构体 os_sem 命名为 OS_SEM,二者等价,只是为了遵循命名规范。

(2) os_sem 结构体定义如下:

```
struct os_sem {
    /* ------------------ 普通成员 ------------------ */
#if OS_OBJ_TYPE_REQ > 0u
    OS_OBJ_TYPE Type;               //类型应该被赋值为 OS_OBJ_TYPE_SEM
#endif
#if OS_CFG_DBG_EN > 0u
    CPU_CHAR *NamePtr;              //指向信号量的名称 (NUL terminated ASCII)
#endif
    OS_PEND_LIST PendList;          //等待此信号量的任务列表
#if OS_CFG_DBG_EN > 0u
    OS_SEM *DbgPrevPtr;
    OS_SEM *DbgNextPtr;
    CPU_CHAR *DbgNamePtr;
#endif
/* ------------------ 特殊成员 ------------------ */
    OS_SEM_CTR                      Ctr;
    CPU_TS                          TS;
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    CPU_INT08U SemID;               //独一无二的 ID 来让第三方调试器和绘图工具使用
#endif
};
```

由定义可见信号量结构体中描述了其类型,名称,此外还维护了一个阻塞在此信号量上的任务队列。

4.2.2 μC/OS-Ⅲ信号量管理函数

信号量管理函数如下:

(1) OSSemCreate():信号量创建函数;

(2) OSSemDel():信号量删除函数;

(3) OSSemPend():申请信号量函数;

(4) OSSemPendAbort():放弃等待信号量函数;

(5) OSSemPost():释放信号量函数;

(6) OSSemSet()：设置信号量函数。

下面简单介绍一下几个重要的函数。

1. OSSemCreate()：信号量创建函数

功能描述：新建一个信号量。

参数描述：

(1) p_sem：是一个 OS_SEM 类型的指针，OS_SEM 即 os_sem，应在应用程序中新建一个 os_sem 实例。

(2) p_name：指向信号量名称的指针，由使用者来为信号量命名。

cnt 是信号量初始值，表示被此信号量保护的资源可以同时被几个任务使用。

(3) p_err：是一个变量指针，指向一个由此函数生成的状态码，其类别如下：

① OS_ERR_NONE：函数成功执行生成信号量任务；

② OS_ERR_CREATE_ISR：从 ISR(Interrupt Server Routine，中断服务函数)调用此函数；

③ OS_ERR_ILLEGAL_CREATE_RUN_TIME：正在调用；

④ OSSafetyCriticalStart()：之后调用此函数；

⑤ OS_ERR_NAME：p_name 是 NULL pointer；

⑥ OS_ERR_OBJ_CREATED：该信号量已经被创建；

⑦ OS_ERR_OBJ_PTR_NULL：p_sem 是 NULL pointer；

⑧ OS_ERR_OBJ_TYPE：p_sem 已经被初始化为一个其他类型的对象。

返回值：none(无)。

```
void OSSemCreate(OS_SEM *p_sem, CPU_CHAR *p_name,
                 OS_SEM_CTR cnt, OS_ERR *p_err){
    CPU_SR_ALLOC();
//操作系统在 Safely_Critical 模式下,严格保护系统资源
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return;
    }
#endif
#ifdef OS_SAFETY_CRITICAL_IEC61508
    if (OSSafetyCriticalStartFlag == DEF_TRUE) {
        *p_err = OS_ERR_ILLEGAL_CREATE_RUN_TIME;
        return;
    }
#endif
#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {   //不允许从中断服务程序中调用
        *p_err = OS_ERR_CREATE_ISR;
```

```
            return;
        }
    #endif
    #if OS_CFG_ARG_CHK_EN > 0u
        if (p_sem == (OS_SEM *)0) {          //使'p_sem'生效
            *p_err = OS_ERR_OBJ_PTR_NULL;
            return;
        }
    #endif
        CPU_CRITICAL_ENTER();
        p_sem->Type = OS_OBJ_TYPE_SEM;       //数据结构标记为信号量
        p_sem->Ctr = cnt;                    //设置信号量的值
        p_sem->TS = (CPU_TS)0;
        p_sem->NamePtr = p_name;             //保存信号量的名称
        OS_PendListInit(&p_sem->PendList);  //初始化任务等待队列
    #if OS_CFG_DBG_EN > 0u
        OS_SemDbgListAdd(p_sem);
    #endif
        OSSemQty++;
    #if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
        TRACE_OS_SEM_CREATE(p_sem, p_name);  //记录事件
    #endif

        OS_CRITICAL_EXIT_NO_SCHED();
        *p_err = OS_ERR_NONE;
    }
```

2. OSSemDel()：信号量删除函数

销毁一个信号量有两种情况，第一，只有在没有任务等待该信号量的时候才删除；第二，不管有没有任务等待该信号量都强制删除。对于第一种情况，用 OS_OPT_DEL_NO_PEND 进行标识，这时，如果确实没有任务在等待该信号量，将删除信号量结构体的实例，然后返回；如果有任务在等待该信号量，直接返回错误标识，以告知调用者删除失败。对于第二种情况，用 OS_OPT_DEL_ALWAYS 进行标识，首先用 OS_PendObjDel()将等待任务表中的所有任务都移走，然后删除信号量。这个时候要判断是否有任务在等待，如果刚刚有任务在等待该信号量(这个判断在移走之前已经做好)，就必须重新调度 OSSched()。所以，如果当前任务不是优先级最高的任务，那么这个函数将会被挂起。

功能描述：删除一个信号量。

参数描述：

(1) p_sem：指向要删除的信号量指针。

(2) opt：确定删除的选项如下：

① OS_OPT_DEL_NO_PEND：删除信号量，当且仅当信号量没有被任务等待；

② OS_OPT_DEL_ALWAYS：删除信号量，即使它被任务等待使用。

(3) p_err：是一个变量指针，指向一个由此函数生成的状态码，其类别如下：

① OS_ERR_NONE：函数成功执行生成信号量；

② OS_ERR_CREATE_ISR：从 ISR 调用此函数；

③ OS_ERR_ILLEGAL_CREATE_RUN_TIME：正在调用；

④ OSSafetyCriticalStart()：之后调用此函数；

⑤ OS_ERR_NAME：p_name 是 NULL pointer；

⑥ OS_ERR_OBJ_CREATED：该信号量已经被创建；

⑦ OS_ERR_OBJ_PTR_NULL：p_sem 是 NULL pointer；

⑧ OS_ERR_OBJ_TYPE：p_sem 已经被初始化为一个其他类型的对象。

返回值：等于 0 表示没有任务在等待信号量或者存在错误；大于 0 表示有一个或多个等待信号量的任务就绪并通知。

注意事项：

(1) 这个函数必须谨慎使用，任务通常会期望信号量的存在必须检查 OSSemPend()的返回码。

(2) OSSemAccept()的调用者将不会知道信号量将要被删除的意图。

(3) 由于被信号量阻塞的任务都是就绪态，要注意在信号量被当作互斥量来使用的应用程序，因为系统的资源将不再会被此信号量保护。

```
#if OS_CFG_SEM_DEL_EN > 0u                          //系统设置信号量可以被删除
OS_OBJ_QTY  OSSemDel (OS_SEM   *p_sem, OS_OPT   opt, OS_ERR   *p_err){
    OS_OBJ_QTY      cnt;
    OS_OBJ_QTY      nbr_tasks;
    OS_PEND_DATA   *p_pend_data;
    OS_PEND_LIST   *p_pend_list;
    OS_TCB         *p_tcb;
    CPU_TS          ts;
    CPU_SR_ALLOC();
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_OBJ_QTY)0);
    }
#endif
#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        //该条件下不允许从中断服务程序中删除信号量
        *p_err = OS_ERR_DEL_ISR;
        return ((OS_OBJ_QTY)0);
    }
#endif
#if OS_CFG_ARG_CHK_EN > 0u
```

```
    if (p_sem == (OS_SEM *)0) {                 //指向信号量的指针变量
        *p_err = OS_ERR_OBJ_PTR_NULL;
        return ((OS_OBJ_QTY)0);
}
switch (opt) {                                  //变量 'opt'
        case OS_OPT_DEL_NO_PEND:
        case OS_OPT_DEL_ALWAYS:
             break;
        default:
            *p_err = OS_ERR_OPT_INVALID;
             return ((OS_OBJ_QTY)0);
}
#endif
#if OS_CFG_OBJ_TYPE_CHK_EN > 0u
    if (p_sem->Type != OS_OBJ_TYPE_SEM) {       //确保信号量一定被创建
        *p_err = OS_ERR_OBJ_TYPE;
        return ((OS_OBJ_QTY)0);
    }
#endif
    CPU_CRITICAL_ENTER();
    p_pend_list = &p_sem->PendList;
    cnt          = p_pend_list->NbrEntries;
    nbr_tasks    = cnt;
    switch (opt) {
        case OS_OPT_DEL_NO_PEND:
             //当且仅当任务等待队列为空时删除信号量
             if (nbr_tasks == (OS_OBJ_QTY)0) {
#if OS_CFG_DBG_EN > 0u
                 OS_SemDbgListRemove(p_sem);
#endif
                 OSSemQty--;
                 OS_SemClr(p_sem);
                 CPU_CRITICAL_EXIT();
                 *p_err = OS_ERR_NONE;
             } else {
                 CPU_CRITICAL_EXIT();
                 *p_err = OS_ERR_TASK_WAITING;
             }
             break;
        case OS_OPT_DEL_ALWAYS:                 //总是删除信号量
             OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
            //得到本地时间戳,使得所有任务得到相同的时间
             ts = OS_TS_GET();
             while (cnt > 0u) {                 //从任务等待列表中删除所有任务
                 p_pend_data = p_pend_list->HeadPtr;
                 p_tcb        = p_pend_data->TCBPtr;
```

```
                    OS_PendObjDel((OS_PEND_OBJ * )((void * )p_sem),
                                  p_tcb,
                                  ts);
                     cnt--;
                  }
#if OS_CFG_DBG_EN > 0u
                  OS_SemDbgListRemove(p_sem);
#endif
                  OSSemQty--;
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
                  TRACE_OS_SEM_DEL(p_sem);               //记录事件
#endif
                  OS_SemClr(p_sem);
                  OS_CRITICAL_EXIT_NO_SCHED();
                   //从等待列表中寻找最高优先级的任务来运行
                  OSSched();
                   * p_err = OS_ERR_NONE;
                  break;
             default:
                  CPU_CRITICAL_EXIT();
                   * p_err = OS_ERR_OPT_INVALID;
                  break;
    }
    return ((OS_OBJ_QTY)nbr_tasks);
}
#endif
```

3. OSSemPend()：申请信号量，相当于 P 操作

功能描述：若信号量可用资源数大于 0，则分配资源，否则挂起任务，等待信号量。

参数描述：

(1) p_se：指向信号量的指针。

(2) timeout：是一个可选的超时周期(in clock ticks)。如果非零，任务将等待指定时间的资源。如果指定 0，在这种情况下任务将一直等待直到指定的信号量或资源可用(或者事件发生)。

(3) opt：这个参数决定用户在信号量可用时是否阻塞，两个状态值如下：

① OS_OPT_PEND_BLOCKING；

② OS_OPT_PEND_NON_BLOCKING。

(4) p_ts：是一个指针变量，它在信号量被释放、终止或删除时接受一个时间戳。如果传进来 NULL 指针(i.e.(CPU_TS *)0)，将得不到此时间戳。即传进一个 NULL 指针是合法的并且表示不需要此时间戳。

(5) p_err：是一个变量指针，指向一个由此函数生成的状态码，其类别如下：

① OS_ERR_NONE：函数成功执行并且任务得到了资源或等待的事件已经发生；

② OS_ERR_OBJ_DEL：信号量 p_sem 被删除；

③ OS_ERR_OBJ_PTR_NULL：信号量 p_sem 是 NULL 指针；

④ OS_ERR_OBJ_TYPE：信号量 p_sem 没有指向一个信号量；

⑤ OS_ERR_OPT_INVALID：调用者为 opt 指定了一个非法值；

⑥ OS_ERR_PEND_ABORT：任务挂起被另一个任务终止；

⑦ OS_ERR_PEND_ISR：函数被从 ISR 中调用并且将引发一个暂停；

⑧ OS_ERR_PEND_WOULD_BLOCK：调用者指定为非阻塞(non-blocking)但信号量不能被获取；

⑨ OS_ERR_SCHED_LOCKED：当调度程序被锁死时调用此函数；

⑩ OS_ERR_STATUS_INVALID：挂起状态非法；

⑪ OS_ERR_TIMEOUT：在指定时间内信号量没有被获取。

返回值：返回信号量计数器的值，当信号量不可获取时返回 0。

```
OS_SEM_CTR  OSSemPend(OS_SEM   * p_sem, OS_TICK   timeout,
                      OS_OPT    opt, CPU_TS    * p_ts, OS_ERR    * p_err){
    OS_SEM_CTR    ctr;
    OS_PEND_DATA  pend_data;
    CPU_SR_ALLOC();
#ifdef OS_SAFETY_CRITICAL
if (p_err == (OS_ERR * )0) {
    #if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
         TRACE_OS_SEM_PEND_FAILED(p_sem);              //记录此事件
#endif
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_SEM_CTR)0);
     }
#endif
#if OS_CFG_CALLED_FROM_ISR_CHK_EN > 0u
     if (OSIntNestingCtr > (OS_NESTING_CTR)0) {        //此条件下不允许从 ISR 中调用
         * p_err  =  OS_ERR_PEND_ISR;
         return ((OS_SEM_CTR)0);
     }
#endif

#if OS_CFG_ARG_CHK_EN > 0u
    if (p_sem == (OS_SEM * )0) {                       //验证参数 p_sem
        * p_err  =  OS_ERR_OBJ_PTR_NULL;
        return ((OS_SEM_CTR)0);
    }
    switch (opt) {                                     //验证参数 opt
        case OS_OPT_PEND_BLOCKING:
        case OS_OPT_PEND_NON_BLOCKING:
             break;
```

```
            default:
                 *p_err = OS_ERR_OPT_INVALID;
                return ((OS_SEM_CTR)0);
        }
#endif
#if OS_CFG_OBJ_TYPE_CHK_EN > 0u
    if (p_sem->Type != OS_OBJ_TYPE_SEM) {          //确认信号量被创建
         *p_err = OS_ERR_OBJ_TYPE;
        return ((OS_SEM_CTR)0);
    }
#endif
    if (p_ts != (CPU_TS *)0) {
        *p_ts = (CPU_TS)0;                          //初始化返回时间戳的值
    }
    CPU_CRITICAL_ENTER();
    if (p_sem->Ctr > (OS_SEM_CTR)0) {              //资源是否可用
        p_sem->Ctr--;                               //若可用,调用任务
        if (p_ts != (CPU_TS *)0) {
            *p_ts = p_sem->TS;                      //获取最后一次释放信号量时的时间戳
        }
        ctr = p_sem->Ctr;
        CPU_CRITICAL_EXIT();
         *p_err = OS_ERR_NONE;
        return (ctr);
    }
    if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) {
        //资源不可用时任务是否想要阻塞挂起?
        ctr = p_sem->Ctr;
        //不阻塞挂起,任务没有加入等待队列中
        CPU_CRITICAL_EXIT();
         *p_err = OS_ERR_PEND_WOULD_BLOCK;
        return (ctr);
    } else {                                        //任务阻塞挂起,任务加入等待队列中
        if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
            //当调度程序锁死,任务不能挂起
            CPU_CRITICAL_EXIT();
             *p_err = OS_ERR_SCHED_LOCKED;
            return ((OS_SEM_CTR)0);
        }
    }
    //锁死调度程序,重新启用中断
    OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();
    OS_Pend(&pend_data,                             //任务等待信号量
            (OS_PEND_OBJ *)((void *)p_sem),
            OS_TASK_PEND_ON_SEM,
            timeout);
```

```
    OS_CRITICAL_EXIT_NO_SCHED();
    OSSched();                              //找到下一个优先级最高的任务来执行
CPU_CRITICAL_ENTER();
      switch (OSTCBCurPtr->PendStatus) {
          case OS_STATUS_PEND_OK:           //得到信号量
               if (p_ts != (CPU_TS *)0) {
                   *p_ts = OSTCBCurPtr->TS;
               }
                *p_err = OS_ERR_NONE;
               break;
          case OS_STATUS_PEND_ABORT:        //表明终止
               if (p_ts != (CPU_TS *)0) {
                   *p_ts = OSTCBCurPtr->TS;
               }
                *p_err = OS_ERR_PEND_ABORT;
               break;
        //表明我们没有在设定时间内得到信号量
        case OS_STATUS_PEND_TIMEOUT:
             if (p_ts != (CPU_TS *)0) {
                 *p_ts = (CPU_TS)0;
             }
              *p_err = OS_ERR_TIMEOUT;
             break;
        case OS_STATUS_PEND_DEL:            //表明这个对象等待的信号量已经被删除
             if (p_ts != (CPU_TS *)0) {
                 *p_ts = OSTCBCurPtr->TS;
             }
              *p_err = OS_ERR_OBJ_DEL;
             break;
        default:
              *p_err = OS_ERR_STATUS_INVALID;
             CPU_CRITICAL_EXIT();
             return ((OS_SEM_CTR)0);
    }
    ctr = p_sem->Ctr;
    CPU_CRITICAL_EXIT();
    return (ctr);
}
```

4. OSSemPost()：释放信号量函数，相当于V操作

功能描述：这个功能释放一个信号量。

参数描述：

(1) p_sem：一个指向信号量的指针。

(2) opt：决定释放执行的类型。

① OS_OPT_POST_1：当最高优先级的任务等待信号量时使其就绪并将信号量的句

柄给此任务；

② OS_OPT_POST_ALL：将信号量释放的消息告知所有等待队列中的任务；

③ OS_OPT_POST_NO_SCHED：不调用任务调度程序；

④ OS_OPT_POST_NO_SCHED：可以和其他任意一个选项加在一起。

(3) p_err：是一个变量指针，指向一个由此函数生成的状态码，其类别如下：

① OS_ERR_NONE：函数成功执行并且信号量成功被释放；

② OS_ERR_OBJ_PTR_NULL：信号量 p_sem 是 NULL 指针；

③ OS_ERR_OBJ_TYPE：信号量 p_sem 没有指向一个信号量；

④ OS_ERR_SEM_OVF：释放操作使信号量计数溢出。

返回值：返回信号量的计数或当出现错误时返回 0。

```
OS_SEM_CTR  OSSemPost(OS_SEM  *p_sem, OS_OPT  opt, OS_ERR  *p_err)
{
    OS_SEM_CTR  ctr;
    CPU_TS      ts;
#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_SEM_CTR)0);
    }
#endif
#if OS_CFG_ARG_CHK_EN > 0u
    if (p_sem == (OS_SEM *)0) {                     //验证参数 p_sem
        *p_err = OS_ERR_OBJ_PTR_NULL;
        return ((OS_SEM_CTR)0);
}
switch (opt) {                                      //验证参数 opt
        case OS_OPT_POST_1:
        case OS_OPT_POST_ALL:
        case OS_OPT_POST_1   | OS_OPT_POST_NO_SCHED:
        case OS_OPT_POST_ALL | OS_OPT_POST_NO_SCHED:
             break;

        default:
#endif
#if OS_CFG_OBJ_TYPE_CHK_EN > 0u
    if (p_sem->Type != OS_OBJ_TYPE_SEM) {           //确认信号量被创建
         *p_err = OS_ERR_OBJ_TYPE;
        return ((OS_SEM_CTR)0);
    }
#endif
    ts = OS_TS_GET();                               //获得时间戳
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
```

```
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {         //看是否是从 ISR 中调用
        OS_IntQPost((OS_OBJ_TYPE)OS_OBJ_TYPE_SEM,   //传递到 ISR 队列中
                    (void        * )p_sem,
                    (void        * )0,
                    (OS_MSG_SIZE)0,
                    (OS_FLAGS    )0,
                    (OS_OPT      )opt,
                    (CPU_TS      )ts,
                    (OS_ERR      * )p_err);
        return ((OS_SEM_CTR)0);
    }
#endif
    ctr = OS_SemPost(p_sem, opt, ts, p_err);           //释放信号量
    return (ctr);
}
```

4.2.3 μC/OS-Ⅲ信号量应用开发

试用信号量模拟生产者-消费者模型。设进程A是生产者，进程B是消费者，系统最多只能同时容纳5个产品，初始成品数为0。当产品不足5时允许进程A生产；当有产品时允许进程B消费，如图4.2所示。

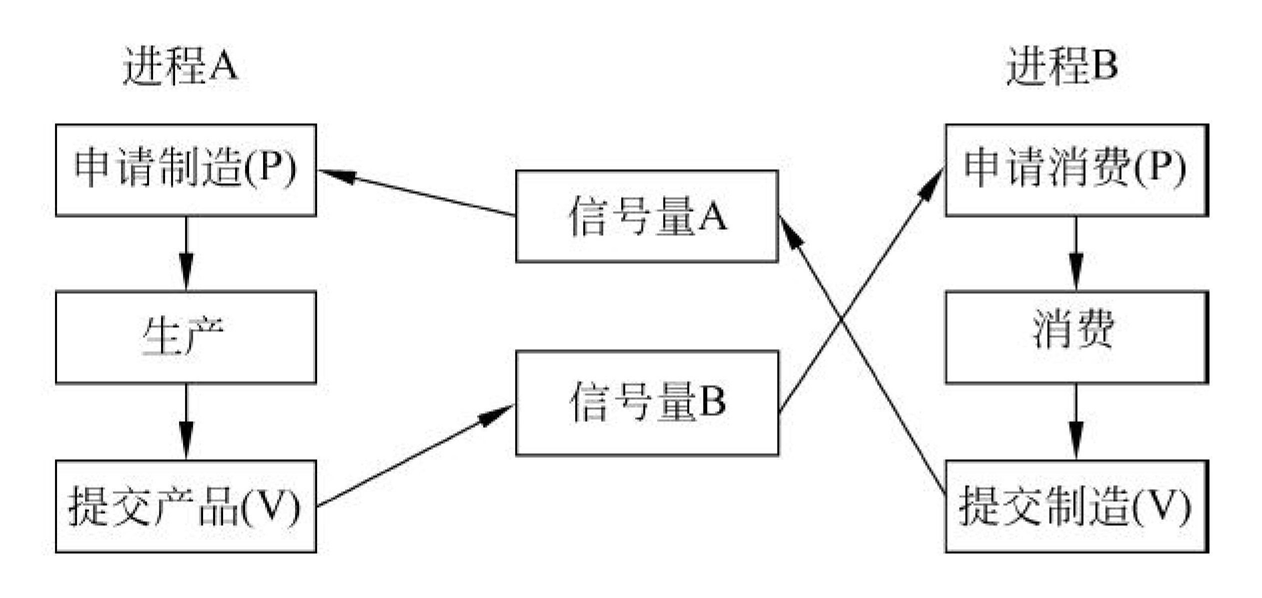


图4.2 生产消费图

```
#include <includes.h>
#define  TASK_STK_SIZE     5
OS_STK          AppStk_Producer[TASK_STK_SIZE];
OS_STK          AppStk_Consumer[TASK_STK_SIZE];
static void App_Producer(void * p_arg);
static void App_Consumer(void * p_arg);
OS_EVENT * sem_full;
OS_EVENT * sem_empty;
OS_EVENT * sem_mutex;
static INT32U food = 0;
void main(int argc, char * argv[])
{
```

```
    OSInit();
    sem_full = OSSemCreate(0);
    sem_empty = OSSemCreate(100);
    sem_mutex = OSSemCreate(1);
    OSTaskCreate(App_Producer,NULL,(OS_STK *)&AppStk_Producer[TASK_STK_SIZE-1],(INT8U)10);
    OSTaskCreate(App_Consumer,NULL,(OS_STK *)&AppStk_Consumer[TASK_STK_SIZE-1],(INT8U)11);
    OSStart();                                   //Start multitasking
}
void App_Producer(void *p_arg)
{
    INT8U err;
    p_arg = p_arg;
    while (TRUE)
    {
        OSSemPend(sem_empty, 0, &err);
        OSSemPend(sem_mutex, 0, &err);
        food++;
        printf("生产者: 食物数量[%03d]\n", food);
        OSSemPost(sem_mutex);
        OSSemPost(sem_full);
        OSTimeDlyHMSM(0, 0, 1, 0);
    }
}
void App_Consumer(void *p_arg)
{
    INT8U err;
    p_arg = p_arg;
    while (TRUE)
    {
        OSSemPend(sem_full, 0, &err);
        OSSemPend(sem_mutex, 0, &err);
        food--;
        printf("消费者: 食物数量[%03d]\n", food);
        OSSemPost(sem_mutex);
        OSSemPost(sem_empty);
        OSTimeDlyHMSM(0, 0, 3, 0);
    }
}
```

4.3 μC/OS-Ⅲ互斥体机制分析

本章介绍的内容是互斥体，又称作互斥信号量，可以看作是信号量的一种特例，因为信号量所维护资源只有一个，即信号量初始值为1。互斥信号量主要用来解决优先级反转问题，也可以作为维护系统重要资源的信号量。例如，某一个挂载在系统上的设备需要被多个

并发的任务来访问时，可以用互斥体来维护临界区，这里不再介绍互斥体的此用途。

信号量主要解决优先级反转问题，优先级反转指的是一种较低优先级任务在较高优先级任务之前执行的现象。当低优先级的任务占有互斥资源时，高优先级的任务到来时由于资源被占用阻塞，此时如果来了中优先级的任务，将抢占低优先级的任务，导致中优先级任务在高优先级任务之前完成，高优先级任务一直被阻塞，发生优先级反转。

而互斥体解决该问题的方法是优先级继承。如果当前占有互斥体的任务优先级低于等待的任务，那么将当前任务等级暂时提高到与新任务相同的优先级。

1. 优先级反转举例

假设有三个任务 a、b 和 c，a 优先级高于 b，b 优先级高于 c，a 和 c 都需要访问一个共享资源 s，保护该资源的信号量为互斥信号量，如图 4.3 所示。

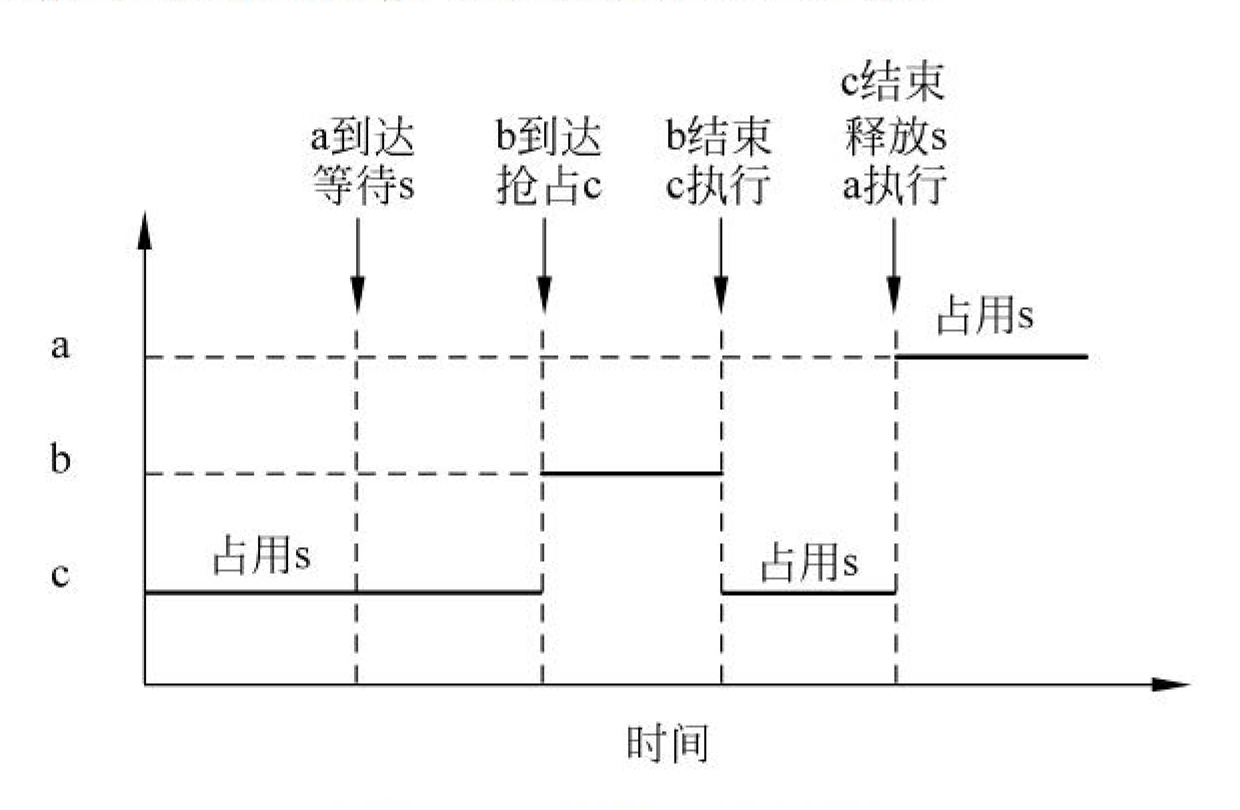


图 4.3 优先级反转举例

假设当前任务 c 申请了信号量访问 s，且还没有释放，此时任务 a 开始运行，那么 a 就会剥夺 c 的运行而运行 a，当 a 去访问资源 s 的时候，因为得不到信号量，所以必须释放以等待信号量，任务 c 得以重新运行，到这里流程都是正常的，信号量的设计也是为了满足这个功能，但是，当任务 c 在运行并准备释放信号量的时候，任务 b 开始运行，那么任务 b 就要剥夺任务 c 的运行，这时系统就只有 b 在运行，a 能打断 b 的运行但是需要信号量，又因为 c 优先级比较低得不到运行，这样，a 就只能等到 b 运行完主动释放使用权才能得到运行。

到这里问题就发生了，优先级比较高的 a 在优先级比较低的 b 运行时无法抢占，可剥夺性内核却剥夺不了 b 的运行，系统故障，这种故障极大地降低了系统的实时性。

2. 实现机制

μC/OS-Ⅲ为了解决上述问题，在互斥信号量中引入了提升优先级的方法，基本思想是，让当前获得互斥信号量的任务优先级短暂提升到系统可以接受的最大优先级，尽量让该任务快速地完成并释放信号量，释放之后再恢复为任务原来的优先级别。

主要看优先级提升部分的代码，可以猜测优先级的提升应该是在一个任务获取了信号量之后完成的，即在 ospendxxx 函数里面。查看 OSMutexPend 函数，结果如下。

```
pip = (INT8U)(pevent->OSEventCnt >> 8u);
if ((INT8U)(pevent->OSEventCnt&OS_MUTEX_KEEP_LOWER_8)
                                == OS_MUTEX_AVAILABLE) {
   pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
   pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
   pevent->OSEventPtr  = (void *)OSTCBCur;
   if (OSTCBCur->OSTCBPrio <= pip) {
     OS_EXIT_CRITICAL();
     *perr = OS_ERR_PIP_LOWER;
   } else {
     OS_EXIT_CRITICAL();
     *perr = OS_ERR_NONE;
   }
}
```

pip变量保存在OSEventCnt中，当创建信号量时，就会给定这个值，这个值也就是系统能够将等待该互斥信号量的任务提升的最高优先级。当一个任务请求信号量时，如果有信号量空余，将当前请求信号量的任务优先级放到OSEventCnt的低八位中。

```
if (OSTCBCur->OSTCBPrio <= pip)
```

如果当前请求信号量的任务优先级高于最高提升优先级(数值上低于)，直接运行，没必要提升优先级，否则，就要进行下面的操作。

```
mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
ptcb = (OS_TCB *)(pevent->OSEventPtr);
if (ptcb->OSTCBPrio > pip) {
    if (mprio > OSTCBCur->OSTCBPrio) {
        y = ptcb->OSTCBY;
        if ((OSRdyTbl[y] & ptcb->OSTCBBitX) != 0u) {
        OSRdyTbl[y] &= (OS_PRIO)~ptcb->OSTCBBitX;
        if (OSRdyTbl[y] == 0u) {
        OSRdyGrp &= (OS_PRIO)~ptcb->OSTCBBitY;
        }
        rdy = OS_TRUE;
```

这段代码的意思是当优先级低于最高可提升优先级时，将系统就绪表中原来的ready标志清除掉，接下来操作如下。

```
ptcb->OSTCBPrio = pip;
ptcb->OSTCBY = (INT8U)( ptcb->OSTCBPrio >> 3u);
```

```
ptcb->OSTCBX = (INT8U)(ptcb->OSTCBPrio & 0x07u);
ptcb->OSTCBBitY = (OS_PRIO)(1uL << ptcb->OSTCBY);
ptcb->OSTCBBitX = (OS_PRIO)(1uL << ptcb->OSTCBX);
if (rdy == OS_TRUE) {
      OSRdyGrp| = ptcb->OSTCBBitY;
      OSRdyTbl[ptcb->OSTCBY] | = ptcb->OSTCBBitX;
} else {
    pevent2 = ptcb->OSTCBEventPtr;
    if (pevent2 != (OS_EVENT *)0) {
    pevent2->OSEventGrp| = ptcb->OSTCBBitY;
    pevent2->OSEventTbl[ptcb->OSTCBY] | = ptcb->OSTCBBitX;
     }
}
OSTCBPrioTbl[pip] = ptcb;
```

将当前任务的优先级切换成提升优先级,并改变快速访问就绪表元素的数据,同时修改系统就绪表,将提升优先级任务的新优先级在任务就绪表中设置成就绪,最后,在 tcb 表中对应 pip 的位置,设置为提升了优先级任务的 tcb。这样,任务的优先级就被提升了,系统下一次被调用时,就会按照被提升了优先级任务的新优先级来进行调度。

既然优先级能被提升,那么也应该能被降下来,而降下来应该需要依靠 ospostxxx 在释放信号量时执行,查看 OSMutexPost 代码,可知如下结果。

```
if (OSTCBCur->OSTCBPrio == pip) {
    OSMutex_RdyAtPrio(OSTCBCur, prio);
}
```

即当释放信号量任务的优先级等于互斥信号量最高可提升优先级时,需要通过 OSMutex_RdyAtPrio 函数来将任务的优先级恢复,prio 是从事件中获取的优先级,而提升之前的优先级保存到了 OSEventCnt 的低八位。

```
pevent->OSEventCnt | = OSTCBCur->OSTCBPrio;
```

恢复优先级的函数为 OSMutex_RdyAtPrio,核心代码如下:

```
y = ptcb->OSTCBY;
OSRdyTbl[y] &= (OS_PRIO)~ptcb->OSTCBBitX;
if (OSRdyTbl[y] == 0u) {
        OSRdyGrp &= (OS_PRIO)~ptcb->OSTCBBitY;
}
ptcb->OSTCBPrio = prio;
OSPrioCur = prio;
```

```
#if OS_LOWEST_PRIO <= 63u
ptcb->OSTCBY = (INT8U)((INT8U)(prio >> 3u) & 0x07u);
ptcb->OSTCBX = (INT8U)(prio & 0x07u);
#else
ptcb->OSTCBY = (INT8U)((INT8U)(prio >> 4u) & 0x0Fu);
ptcb->OSTCBX = (INT8U) (prio & 0x0Fu);
#endif
ptcb->OSTCBBitY = (OS_PRIO)(1uL << ptcb->OSTCBY);
ptcb->OSTCBBitX = (OS_PRIO)(1uL << ptcb->OSTCBX);
OSRdyGrp |= ptcb->OSTCBBitY;
OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
OSTCBPrioTbl[prio] = ptcb;
```

上述流程与之前的权限提升过程类似，只是一个换成高优先级一个换成低优先级。先将高优先级任务的任务就绪表中对应的位清除，然后重新设置任务的原始优先级以及当前任务优先级（释放信号量和申请信号量的是同一个任务），然后设置快速访问就绪任务表的数据元素，重新设置任务就绪表，最后将 tcb 数组中对应原始优先级的数据指针设置到指向任务 tcb，这样就实现了任务的恢复。

4.3.1　μC/OS-Ⅲ互斥体管理函数

互斥体管理函数如下：

（1）OSMutexCreate()：互斥体创建函数；

（2）OSMutexDel()：互斥体删除函数；

（3）OSMutexPend()：申请互斥体函数；

（4）OSMutexPendAbort()：放弃等待互斥体函数；

（5）OSMutexPost()：释放互斥体函数；

（6）OSMutexSet()：设置互斥体函数。

互斥体信号量本就是信号量的特化，实现逻辑基本没有改动，其主要操作函数与信号量相比仅仅是对函数操作名和变量名稍作修改并加入了一些解决优先级反转问题的操作（前文有介绍），这里不再介绍，请参考信号量的函数介绍。

4.3.2　μC/OS-Ⅲ互斥体应用开发

应用开发也和信号量的问题类似，只是运行代码的系统允许优先级抢占，且对任务实时性要求高，但这些都是用户不可见的，因此应用开发基本和信号量相同，只需将信号量换为互斥体，在此也不做开发介绍。

4.4 μC/OS-Ⅲ事件标志组机制分析

在一般的任务同步中,使用信号量和消息队列就可以满足要求,但是在复杂的任务同步中,当一个任务需要多个触发条件时,则需要应用到事件标志组。当某一事件发生时,将其对应标志位置为 1,反之置为 0。

信号量和互斥信号量都是用来同步任务对共享资源的访问,防止冲突而设立的。事件标志组是用来同步几个任务,协调它们有序工作而设立的。事件标志组可以看作是一个信号量集,只有同时获取多个信号量时,才能继续执行任务。事件标志组用来标志等待一组事件发生,可以通过编程用一组信号量来替代,但代价很大,所以就有了事件标志组这个数据结构,用来解决此类问题。实际中若想要读取数据,那么要等数据采集更新完成以后再去读才有意义,所以数据采集和数据读取这两个任务也可以用事件标志组来实现。当然,事件标志组不一定只用于两个任务之间,通过对头文件的修改,可以让事件标志组达到 32 位,即可以用事件标志组来协调多个任务的合理运行,达到预期目的。

将事件标志置位或复位是通过把当前事件标志组和新的标志组进行逻辑“与”或逻辑“或”来实现的。为了得到事件标志,也要做类似的逻辑操作。一旦事件标志组中有标志被置位,系统就会检查相应组的挂起队列,如果能满足挂起线程,则该线程就恢复,如图 4.4 所示。

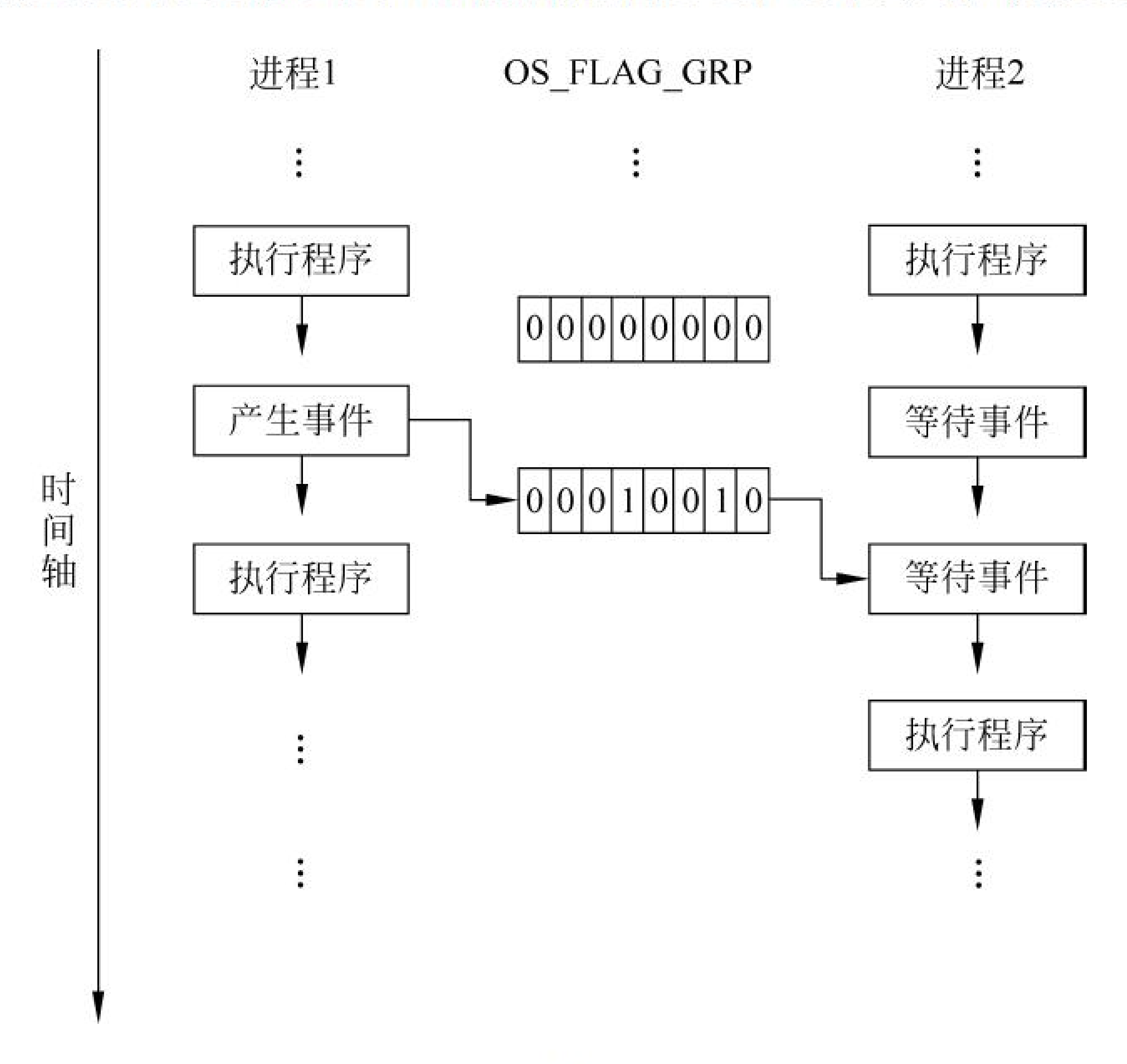


图 4.4 事件标志组机制

事件标志组的结构比其他结构略显复杂。每一个事件标志组都维护着自己的一个等待队列的双向链表。每个事件标志组的节点里面都有一个指针和相应的任务控制块 TCB 一一对应。可以查看相应源代码了解事件标志组的具体实现方法。

事件标志可以被任何线程置位和复位，也可以被任何程序查看，线程可以因等待一组事件标志被置位而挂起。每个事件标志由一个位代表，每 32 个事件标志被安排在一组。

4.4.1 μC/OS-Ⅲ事件标志组关键数据结构

```
typdef  os_flag_grp  OS_FLAG_GRP
struct  os_flag_grp {                              //事件标志组
/* ------------------ 基本成员 ------------------ */
#if OS_OBJ_TYPE_REQ > 0u
      OS_OBJ_TYPE     Type;                        //值应该被设置为 OS_OBJ_TYPE_FLAG
#endif
#if OS_CFG_DBG_EN > 0u
      CPU_CHAR     *NamePtr;               //指向事件标志组名称的指针(非 NULL 的 ASCII 字码)
#endif
     OS_PEND_LIST     PendList;                    //指向事件标志组上的任务等待队列
#if OS_CFG_DBG_EN > 0u
     OS_FLAG_GRP           *DbgPrevPtr;
     OS_FLAG_GRP           *DbgNextPtr;
     CPU_CHAR              *DbgNamePtr;
#endif
/* ------------------ 特殊成员 ------------------ */
     OS_FLAGS       Flags;                         //8, 16 or 32 bit 的标志位
     CPU_TS          TS;                           //当最后一个事件发生时的时间戳
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
     CPU_INT32U    FlagID;                         //为第三方编译器和绘图工具提供的唯一标识
#endif
};
```

由其定义可知事件标志组成员是由链表组织的，且链表是双向的。用一个 8bit、16bit 或 32bit 的数据来存储事件标志，事件发生则对应位置 1。

4.4.2 μC/OS-Ⅲ事件标志组管理函数

有时一个任务可能需要和多个事件同步，这就需要使用事件标志组。事件标志组与任务之间有“或”同步和“与”同步两种同步机制。当任何一个事件发生，任务都被同步的同步机制是“或”同步；需要所有的事件都发生任务才会被同步的同步机制是“与”同步。

(1) 在 μC/OS-Ⅲ中事件标志组是 OS_FLAG_GRP，这在 os.h 文件中定义。事件标志组中也包含了一串任务，这些任务都在等待着事件标志组中的部分(或全部)事件标志被置 1 或清零。在使用之前，必须创建事件标志组。

(2) 任务和 ISR(中断服务程序)都可以发布事件标志，但是只有任务可以创建或删除事件标志组以及取消其他任务对事件标志组的等待。

(3) 任务可以通过调用函数 OSFlagPend()等待事件标志组中任意个事件标志。调用

函数 OSFlagPend()时可以设置一个超时时间，如果过了超时时间请求的事件还没有被发布，那么任务就会重新进入就绪状态。

(4) 可以设置同步机制为“或”同步或“与”同步。

μC/OS-Ⅲ中关于事件标志组的 API 函数如下，一般情况下只使用 OSFlagCreate()、OSFlagPend()、OSFlagPost()这三个函数。

事件标志组主要操作函数如下：

(1) OSFlagCreat()：新建事件标志组；

(2) OSFlagDel()：删除事件标志组；

(3) OSFlagPend()：申请使用事件标志组；

(4) OSFlagPost()：释放事件标志组；

(5) OSFlagPendGetFlagsRdy()：内部函数，用于制造任务就绪因为所需的事件标志位设置；

(6) OSFlagPendAbsort()：放弃等待事件标志组。

1. 创建事件标志组

在使用事件标志组之前，需要调用函数 OSFlagCreate()创建一个事件标志组。

(1) p_grp：指向事件标志组，事件标志组的存储空间需要应用程序进行实际分配，可以按照下面的例子来定义一个事件标志组。

```
OS_FLAG_GRP EventFlag; EventFlag;EventFlag;
```

(2) p_name：事件标志组的名称。

(3) flags：定义事件标志组的初始值。

(4) p_err：用来保存调用此函数后返回的错误码。

2. 等待事件标志组

等待一个事件标志组需要调用函数 OSFlagPend()。

(1) OSFlagPend()允许将事件标志组里的“与或”组合状态设置成任务的等待条件。任务等待的条件可以是标志组里任意一个置位或清零，也可以是所有事件标志位都置位或清零。如果任务等待的事件标志组不满足设置的条件，那么该任务被置位挂起，直到事件标志组满足条件或已到指定的超时或事件标志被删除或另一个任务终止该挂起状态。

(2) p_grp：指向事件标志组。

(3) flags：bit 序列，任务需要等待事件标志组的哪个位则把相应位置 1，根据设置这个序列可以是 8bit、16bit 或者 32bit。比如任务需要等待事件标志组的 bit0 和 bit1 时(无论是等待置位还是清零)，flag 的值为 0X03。

(4) timeout：指定等待事件标志组的超时时间(时钟节拍数)，如果在指定的超时间内所等待的一个或多个事件没有发生，那么任务恢复运行状态。如果此值设置为 0 则任务将一直等待下去，直到一个或多个事件发生。

(5) opt：决定任务等待的条件是所有标志置位，所有标志清零，任意一个标志位置位还

是任意一个标志位清零，具体的定义如下：

① OS_OPT_PEND_FLAG_CLR_ALL：等待事件标志组所有的位清零；

② OS_OPT_PEND_FLAG_CLR_ANY：等待事件标志组中任意一个标志清零；

③ OS_OPT_PEND_FLAG_SET_ALL：等待事件标志组中所有的位置位；

④ OS_OPT_PEND_FLAG_SET_ANY：等待事件标志组中任意一个标志置位。

调用上面四个选项时还可以搭配以下三个选项：

① OS_OPT_PEND_FLAG_CONSUME：设置是否继续保留该事件标志的状态；

② OS_OPT_PEND_NON_BLOCKING：标志组不满足条件时不挂起任务；

③ OS_OPT_PEND_BLOCKING：标志组不满足条件时挂起任务。

这里应该注意选项 OS_OPT_PEND_FLAG_CONSUME 的使用方法，如果希望任务等待事件标志组的任意一个置位，并在满足条件后将对应位清零那么就可以搭配使用选项 OS_OPT_PEND_FLAG_CONSUME。

(6) p_ts：指向一个时间戳，记录了发送、终止和删除事件标志组的时刻，如果为这个指针赋值 NULL，则函数的调用者将不会收到时间戳。

(7) p_err：用来保存调用此函数后返回的错误码。

3. 向事件标志组发布标志

调用函数 PSFlagPost()可以对事件标志组置位或者清零。一般情况下，需要进行置位或者清零的标志由一个掩码确定(参数 flags)。OSFlagPost()修改完事件标志后，将检查并使那些等待条件已经满足的任务进入就绪态。该函数可以对已经置位或清零的标志进行重复置位和清零操作。

(1) p_grp：指向事件标志组。

(2) flags：决定对哪些位清零和置位，当 opt 参数为 OS_OPT_POST_FLAG_SET 时，参数 flags 中置位的位就会在事件标志组对应的位被置位。当 opt 为 OS_OPT_POST_FLAG_CLR 时，参数 flags 中置位的位在事件标志组中对应的位将被清零。

(3) opt：决定对标志位的操作，有以下两种选项：

① OS_OPT_POST_FLAG_SET：对标志位进行置位操作；

② OS_OPT_POST_FLAG_CLR：对标志位进行清零操作。

(4) p_err：保存调用此函数后返回的错误码。

4.4.3　μC/OS-Ⅲ事件标志组应用开发

简单的 OSFLAGPEND()与 OSFLAGPOST()函数应用如下。

显示：LCD1602 和两个 LED 灯。

TASK1()：建一个 OSFlagPend()若等待事件标志没有发生，该函数挂起任务，若事件标志发生，则 LCD 显示一个值，LED 灯每隔一秒闪一次。

TASK2()：OSFlagPost()向任务 TASK1 发送一个信号量。

TASK3()：OSFlagPost()向任务 TASK1 发送一个信号量，具体看代码。

```
static void Task1(void * pdata)
{
    INT8U error;
    INT8U i = 0;
    pdata = pdata;
    while(1){
      //若等待事件标志没有发生则该函数挂起任务
      OSFlagPend(
         Sem_F,                          //请求信号量集
         (OS_FLAGS)3,                    //请求第 0 位和第 1 位信号
         OS_FLAG_WAIT_SET_ALL,           //都置为 1 时为有效否则任务挂在这里,
                                         //无限等待直到收到为止
          &error);                       //OSFLAGPEND 收到有效信号后在 LCD 显示字符串,
                                         //两个 LED 闪烁,可以根据代码而定
      write_com(0x80 + 10);
      while(table1[i] != '\0')
      {
           write_dat(table1[i]);
           i++;
       }
       GPIO_ResetBits(GPIOD, GPIO_Pin_15);
       GPIO_ResetBits(GPIOD, GPIO_Pin_13);
       OSTimeDlyHMSM(0,0,1,0);           //任务挂起 1 秒,否则优先级低的任务没机会执行
       GPIO_SetBits(GPIOD, GPIO_Pin_15);
       GPIO_SetBits(GPIOD, GPIO_Pin_13);
       OSTimeDlyHMSM(0,0,1,0);           //让两个 LED 每秒闪一次
    }
}
static void Task2(void * pdata)
{
    INT8U error;
    pdata = pdata;
    while(1)
     {
      OSFlagPost(                        //发送信号量集
           Sem_F,
           (OS_FLAGS)2,                  //给第 1 位发信号
           OS_FLAG_SET,                  //信号量置 1
           &error);
           OSTimeDlyHMSM(0, 0, 1, 0);    //等待 1 秒
     }
}
static void Task3(void * pdata)
{
    INT8U error;
    pdata = pdata;
```

```
    while(1){
        //在执行此函数时发生任务切换去执行 TASK1,在 OSFLAGPOST 中发生任务切换
        OSFlagPost(                              //发送信号量集
             Sem_F,
             (OS_FLAGS)1,                        //给第 1 位发信号
             OS_FLAG_SET,                        //信号量置 1
             &error);
             OSTimeDlyHMSM(0, 0, 1, 0);          //等待 1 秒
     }
}
```

main 函数大致如下：

```
void main(void){
#if (OS_TASK_NAME_SIZE > 14) && (OS_TASK_STAT_EN > 0)
INT8U err;
#endif
     //目标板初始化
     Target_Init();
     lcd_init();
     OSInit();
     //设置空闲任务名称
#if OS_TASK_NAME_SIZE > 14
OSTaskNameSet(OS_TASK_IDLE_PRIO, "uC/OS - Ⅱ Idle", &err);
#endif
//设置统计任务名称
#if (OS_TASK_NAME_SIZE > 14) && (OS_TASK_STAT_EN > 0)
OSTaskNameSet(OS_TASK_STAT_PRIO, "uC/OS - Ⅱ Stat", &err);
#endif
      Sem_F = OSFlagCreate(0,&error);
      //用任务建立任务
      OSTaskCreateExt(APP_TaskStart, //void ( * task)(void * pd) 任务首地址
      (void * )0, //void * pdata 数据指针
      &APP_TaskStartStk[APP_TASK_START_STK_SIZE - 1],   //OS_STK * ptos 指向任务堆栈栈顶
                                                        //的指针
      (INT8U)APP_TASK_START_PRIO,                 //INT8U prio 任务优先级
      (INT16U)APP_TASK_START_ID,                  //INT16U id 任务的 ID 号
      &APP_TaskStartStk[0],                       //OS_STK * pbos 指向任务堆栈栈底的指针
      (INT32U)APP_TASK_START_STK_SIZE,            //INT32U stk_size 堆栈容量
      (void * )0,                                 //void * pnext 数据指针
      OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR); //INT16U opt 设定 0 STaskCreateExt 的选项
      OSStart();
}
```

上述例子中，TASK1()和 OSFLGAPEND()需要第 0、1 位都置位时才有效，任务刚进来时不满足即被挂起，等待 OSFLGAPOST()把相应位置 1。TASK2、TASK3 分别把第 0、1 位置位。即等到执行完 TASK3 时，TASK1()有效。

TASK1()如下时即为无等待获取，当信号量不满足，任务也不挂起，即 TASK2()，TASK3()不给 OSFLAGCCEPT()发送信号，TASK1()仍然执行。在本例中，LCD，LED 显示正常。

```
static void Task1(void * pdata)
{
     INT8U error;
     INT8U i = 0;
     pdata = pdata;
     while(1){
        //若等待事件标志没有发生该函数并不挂起该任务
        OSFlagAccept(                          //请求信号量集
               Sem_F,
               (OS_FLAGS)3,                   //请求第 0 位和第 1 位信号
               OS_FLAG_WAIT_SET_ALL,          //第 0 位和第 1 位信号都为 1 为有效
               &error);
             //OSFLAGPEND 收到有效信号后在 LCD 显示字符串,两个 LED 闪烁,可以根据代码而定
        write_com(0x80 + 10);
        while(table1[i] != '\0'){
            write_dat(table1[i]);
            i++;
       }
       GPIO_ResetBits(GPIOD, GPIO_Pin_15);
       GPIO_ResetBits(GPIOD, GPIO_Pin_13);
       OSTimeDlyHMSM(0,0,1,0);                 //任务挂起 1 秒,否则优先级低的任务没机会执行
       GPIO_SetBits(GPIOD, GPIO_Pin_15);
       GPIO_SetBits(GPIOD, GPIO_Pin_13);
       OSTimeDlyHMSM(0,0,1,0);                 //让两个 LED 每秒闪一次
    }
}
```

还可以用 OSFLAGQUERY()来查询事件标志组的状态。根据状态来执行所期望的代码，使用起来很方便。

```
static void Task1(void * pdata)
{
    INT8U error;
    INT8U i = 0,j = 0,k = 0,Flags;
    pdata = pdata;
    while(1){
        Flags = OSFlagQuery(                   //查询事件标志组的状态
```

```
            Sem_F,
            &error);
            switch(Flags)
            {
            case 1:
                 write_com(0x80 + 10);
                 while(table1[i] != '\0'){
                 write_dat(table1[i]);
                 i++;
                 }
                 break;
             case 2:
                  write_com(0x80 + 0x40);
                  while(table2[j] != '\0'){
                      write_dat(table2[j]);
                      j++;
                  }
                  break;
              case 3:
                  write_com(0x80 + 0x40 +  8) ;
                  while(table3[k] != '\0'){
                       write_dat(table3[k]);
                       k++;
                   }
                   break;
             }
        OSTimeDlyHMSM(0, 0, 1, 0); //等待 2 秒
        }
}
```

4.5 μC/OS-Ⅲ消息队列

消息队列,顾名思义,就是一个队列,它是用来收发信息的。

(1) 创建消息队列就相当于创建了一个数组,在创建消息队列时需要给一个值来确定消息队列的长度,这就相当于确定了数组的长度。

(2) 在创建消息队列时,内部是没有消息的,相当于是数组是空的,没有进行写数据。

(3) 发送消息相当于是给数组里写进数据。

(4) 接收消息相当于是对数组的数据进行提取,同时对数组进行清空处理。

一个消息传递的例子如图 4.5 所示。

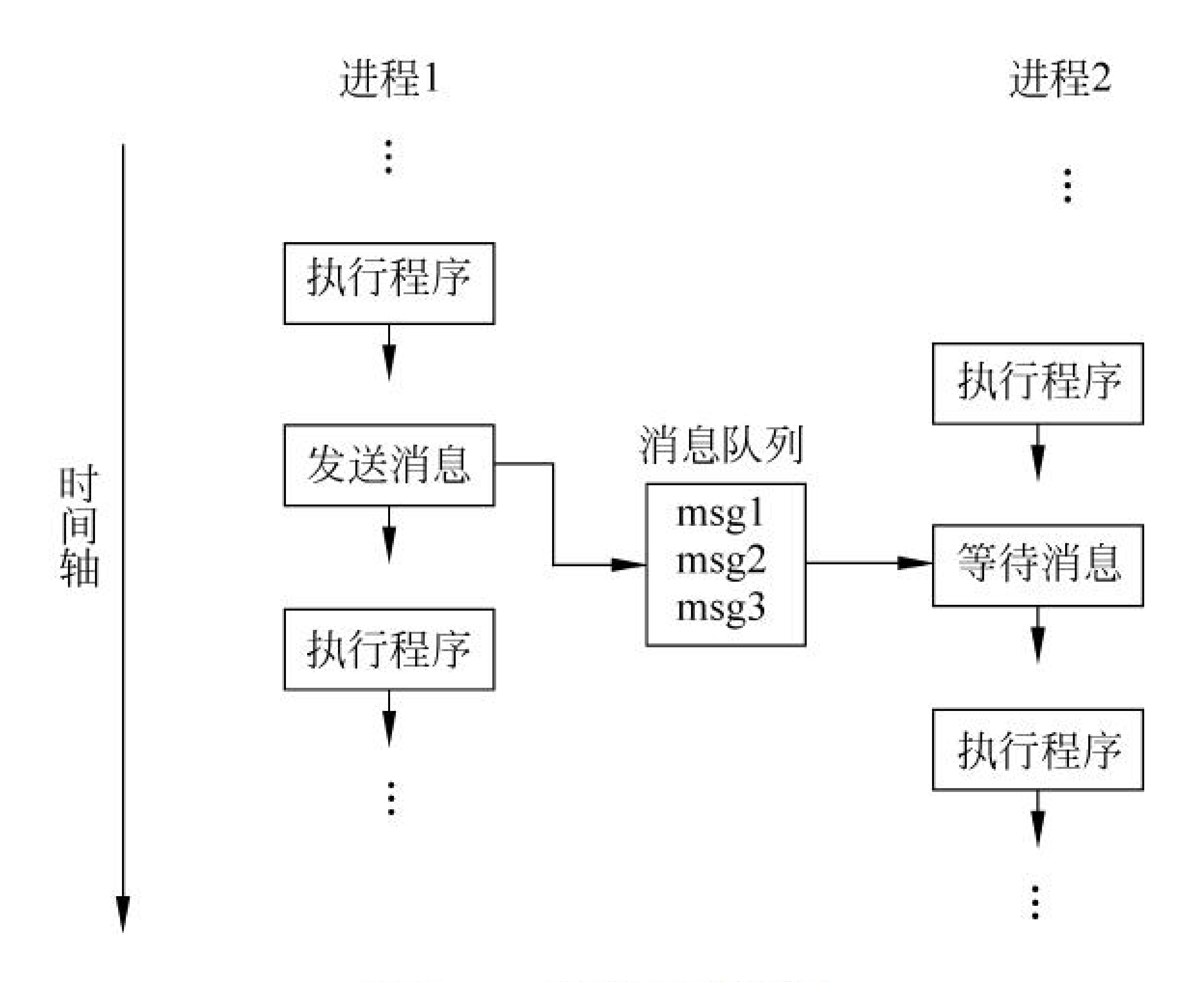


图 4.5 消息队列机制

4.5.1 μC/OS-Ⅲ消息队列数据结构

下面介绍消息队列的主要数据结构。

(1) 消息控制块是消息队列的操作单元,消息控制块 OS_MSG(即 os_msg)如下:

```
struct  os_msg {                          //MESSAGE CONTROL BLOCK
    OS_MSG       * NextPtr;               //指向下一个 OS_MSG
    void     * MsgPtr;                    //实际存储的信息
    OS_MSG_SIZE    MsgSize;               //信息的大小(单位为 bit)
    CPU_TS           MsgTS;               //信息存储进来时的时间戳
};
typedef   struct   os_msg                  OS_MSG;
```

(2) OS_MAG_Q(即 os_mg_q)如下:

```
struct  os_msg_q {  //OS_MSG_Q
    OS_MSG      * InPtr;                     //指向下一个插入消息队列中的消息控制块
    OS_MSG      * OutPtr;                    //指向下一个从消息队列中提取的消息控制块
    OS_MSG_QTY  NbrEntriesSize;              //消息队列最大消息控制块容量
    OS_MSG_QTY  NbrEntries;                  //消息队列当前所存储消息控制块数量
#if OS_CFG_DBG_EN > 0u
    OS_MSG_QTY  NbrEntriesMax;               //队列的消息控制块个数达到最大值
#endif
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    CPU_INT32U   MsgQID;                     //Unique ID for third - party debuggers and tracers.
#endif
};
typedef   struct   os_msg_q    OS_MSG_Q;
```

4.5.2　μC/OS-Ⅲ消息队列操作函数

消息队列函数功能简介如下：

（1）OS_MsgQFreeAll()：释放消息队列中的所有消息；

（2）OS_MsgQGet()：得到消息队列中的消息；

（3）OS_MsgQInit()：消息队列初始化；

（4）OS_MsgQPut()：向消息队列中存入消息。

常用操作函数如下：

（1）void OS_MsgQPut(OS_MSG_Q *p_msg_q,void *p_void,OS_MSG_SIZE msg_size,OS_OPT opt,CPU_TS ts,OS_ERR *p_err)；

此函数的功能是向消息队列中存入消息。参数 p_msg_q 是一个指向消息队列的指针；p_void 是一个指向要存入的真实消息的指针；msg_size 是消息的大小；如果 opt 值为 OS_OPT_POST_FIFO 则队列是先进先出的，如果值为 OS_OPT_POST_LIFO 则队列是先进后出的；ts 是消息存入时的时间戳；p_err 是一个返回标记，其值为 OS_ERR_Q_MAX 说明队列是满的，值为 OS_ERR_MSG_POOL_EMPTY 说明没有消息控制块可使用(注：系统开始时事先生成了一个消息控制块池，这样在使用时不用新生成消息控制块，直接向池中申请一个使用即可)，值为 OS_ERR_NONE 说明消息存入了队列中。此函数的返回值无意义。

（2）void *OS_MsgQGet (OS_MSG_Q *p_msg_q,OS_MSG_SIZE *p_msg_size,CPU_TS *p_ts,OS_ERR *p_err)；

此函数的功能是从消息队列中取出消息。参数 p_msg_q 是一个指向消息队列的指针；p_msg_size 是消息的大小；p_ts 是消息存入时的时间戳；p_err 是一个返回标记，其值为 OS_ERR_Q_EMPTY 说明队列是空的，值为 OS_ERR_NONE 说明消息存入了队列中。此函数的返回值则是指向取出消息的指针。

4.5.3　μC/OS-Ⅲ消息队列应用举例

创建一个可以容纳 10 个消息的全局消息队列。

```
OS_Q        Main_Task_Q;
OSQCreate(&Main_Task_Q,"Main_Task_Q",10,&err);
```

1. 挂起

将任务挂起等待消息，以阻塞方式等待消息到来。

```
OS_MSG_SIZE nMsgSize = 0;
u8 *pMsg = NULL;    //u8 为 unsigned char
CPU_TS nMsgTS;
OS_ERR err;
pMsg = (u8 *)OSQPend(&Main_Task_Q,0,OS_OPT_PEND_BLOCKING,&nMsgSize,&nMsgTS,&err);
```

2. 发送

在一个任务中向另外一个任务发送消息，以FIFO方式放入消息队列，消息内容为“Hello_uC/OS-Ⅲ”。

```
OSQPost(&Main_Task_Q," Hello_uC/OS - Ⅲ ",sizeof("Hello_uC/OS - Ⅲ "),OS_OPT_POST_FIFO,
&err);
```

习题

1. 操作系统的同步机制是什么？为什么需要同步机制？
2. μC/OS-Ⅲ的同步机制有哪些？
3. 请详细描述μC/OS-Ⅲ的信号量机制。
4. 信号量机制的结构和管理函数有哪些？
5. 请详细描述μC/OS-Ⅲ的互斥体机制。
6. 互斥体机制的结构和管理函数有哪些？
7. 请详细描述μC/OS-Ⅲ的事件标志组机制。
8. 事件标志组机制的结构和管理函数有哪些？
9. 请详细描述μC/OS-Ⅲ的消息队列机制。
10. 消息队列机制的结构和管理函数有哪些？

第 5 章

中 断 管 理



5.1 μC/OS-Ⅲ中断机制

中断是计算机中十分重要的组成部分，现代计算机毫无例外地都要采用中断技术。同样地，μC/OS-Ⅲ也提供了中断的支持。中断是指在程序运行过程中，响应内部或外部异步事件的请求，终止当前任务，而去处理异步事件所要求任务的过程。中断服务程序（ISR）是响应中断请求从而执行的程序。中断向量是中断服务程序的入口地址，即存储中断服务程序内存地址的首单元。

中断是操作系统中的一种硬件机制，当发生一个异步事件时，中断机制将负责向 CPU 传达该事件的发生，在 CPU 确认后，保存当前任务的上下文，将其部分或全部寄存器入栈保存，并跳转到负责处理该异步事件的中断服务程序。中断服务程序结束后，进行任务调度。如果存在比被中断任务优先级更高的任务，那么该任务进入就绪状态，等待执行；否则，恢复先前被中断任务的上下文，任务进入就绪态，等待执行。

中断处理的实时性通常比轮询要好，经过中断，微处理器可以在外部事件发生时立刻进行处理，而不需要连续轮询该事件是否发生。在执行中断处理代码之前，通常需要将处理器的寄存器保存入栈。微处理器能够通过特殊的指令来关闭和允许特定的中断请求。关闭中断则会增加中断处理过程的延迟，还可能导致后续中断请求的丢失。因此，在实时系统中，应尽量减少关中断的时间。很多微处理器支持中断嵌套，所谓中断嵌套就是指在处理中断过程中，可以响应新的中断请求，这样，处理器可以及时响应更加重要的中断。中断嵌套如图 5.1 所示。

实时多任务内核的一个重要指标是中断关闭总时间。临界段代码是操作系统在处理时不可分割的代码，一旦这部分代码开始执行，则不允许任何中断打扰。所以实时多任务内核在运行临界段代码之前会关闭中断，在临界段代码运行完后重新打开中断。关闭中断的时间越长，系统的中断等待时间就越长。

中断延迟时间是指从硬件中断发生到开始执行中断处理程序第一条指令之间的这段时间。

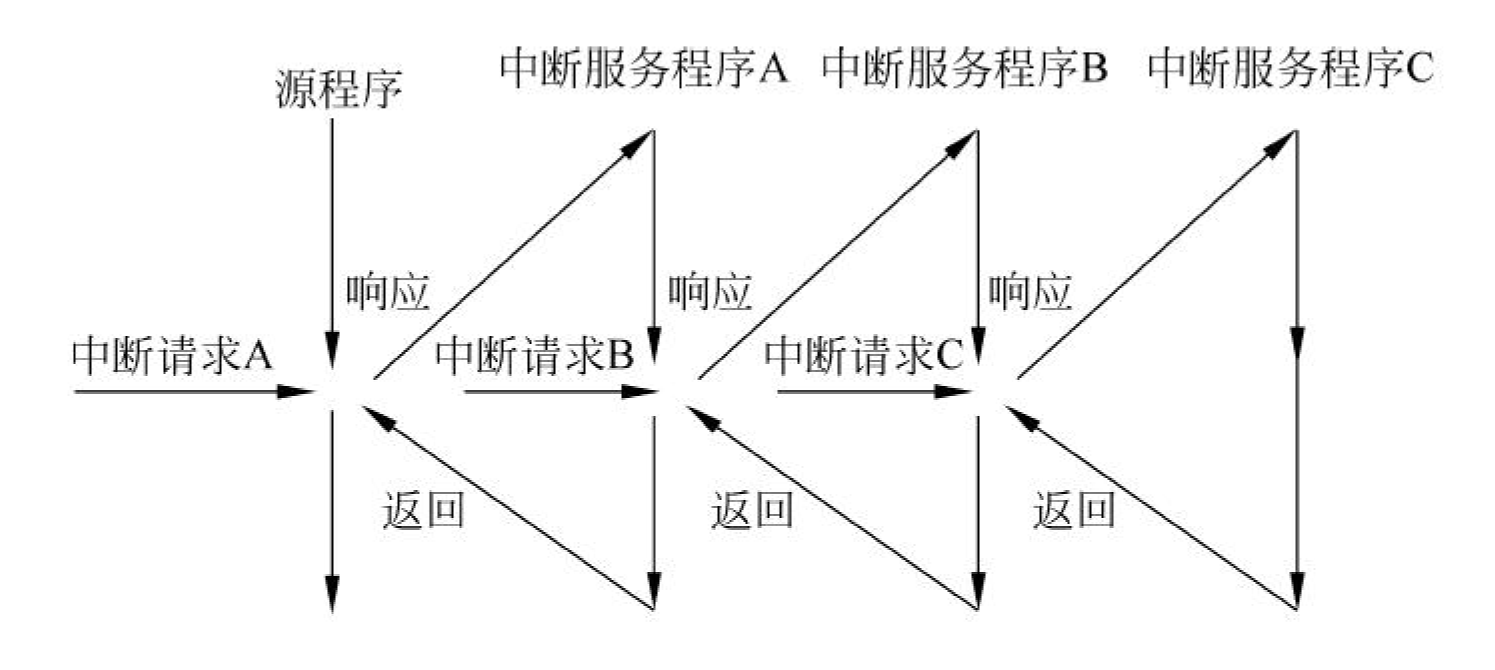


图 5.1　中断嵌套示意图

中断响应时间是指从中断被识别到对应的中断处理代码开始执行的时间。中断响应时间包括中断延迟时间、保存寄存器内容(保护现场)的时间和跳转到中断服务子程序的时间。

中断恢复时间是指从中断代码执行完毕，到被中断的任务代码或由于中断处理而进入就绪状态的更高优先级任务代码开始执行之间的时间。中断恢复时间包括恢复寄存器内容(恢复现场)的时间和执行中断返回指令的时间。

任务等待时间是指从中断发生到任务代码重新开始执行的时间。

μC/OS-Ⅲ响应中断的过程如图 5.2 所示。

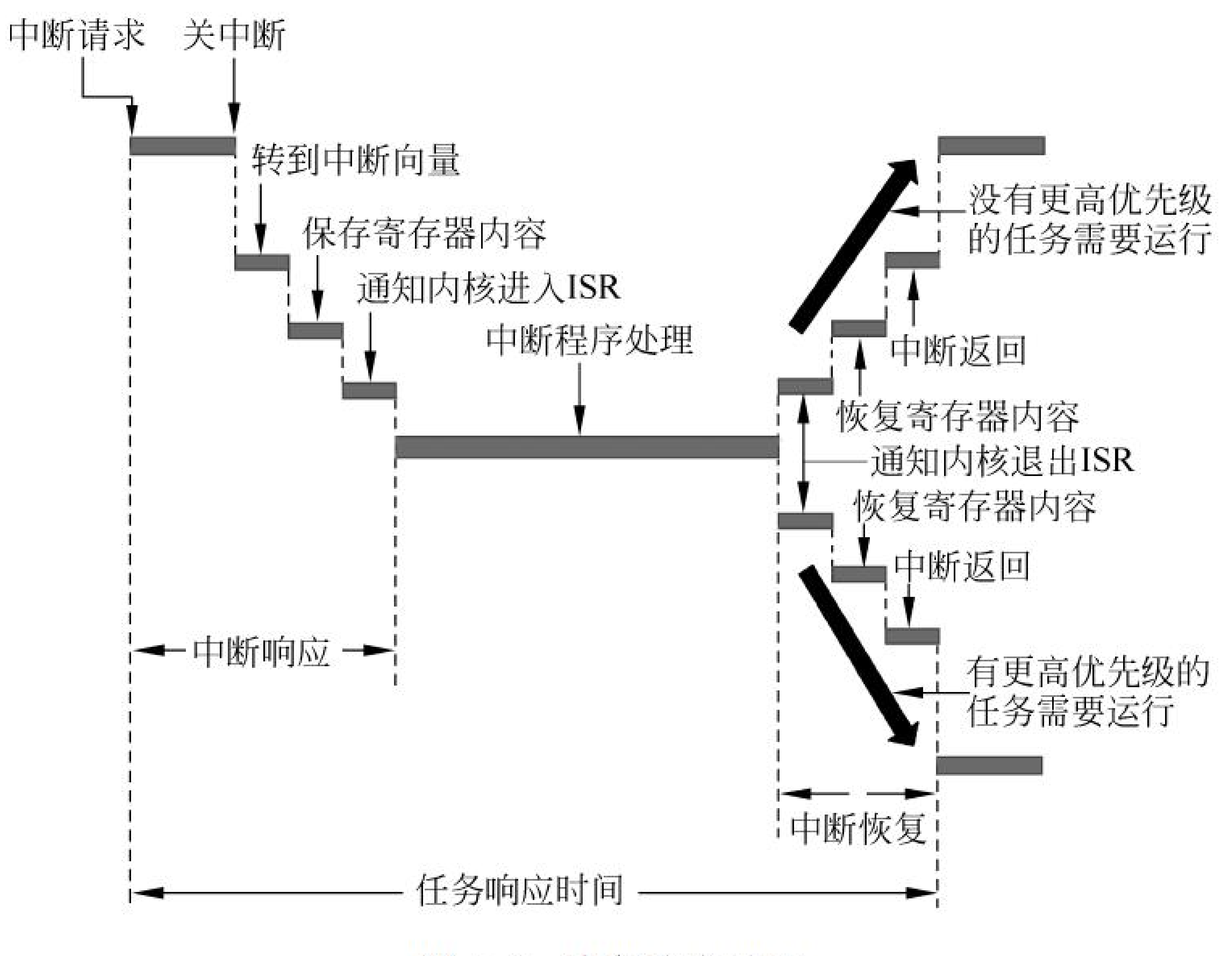


图 5.2　中断响应过程

5.2 CPU 中断处理

所有的中断请求信号通过中断控制器接收，当前大多数 CPU 框架都能接收处理多个中断源请求。例如：用户点击鼠标，敲击键盘，UART 接收到字符，以太网控制器接收到数据帧，DMA 控制器完成传输等都会触发中断。中断控制器能够同时接收多个中断请求，允许为每个中断请求设置优先级，并将优先级最高的中断请求的服务程序地址直接传递给 CPU。CPU 保存当前任务寄存器的内容，然后跳转到中断向量处，执行中断处理程序。中断控制器的运行过程如图 5.3 所示。

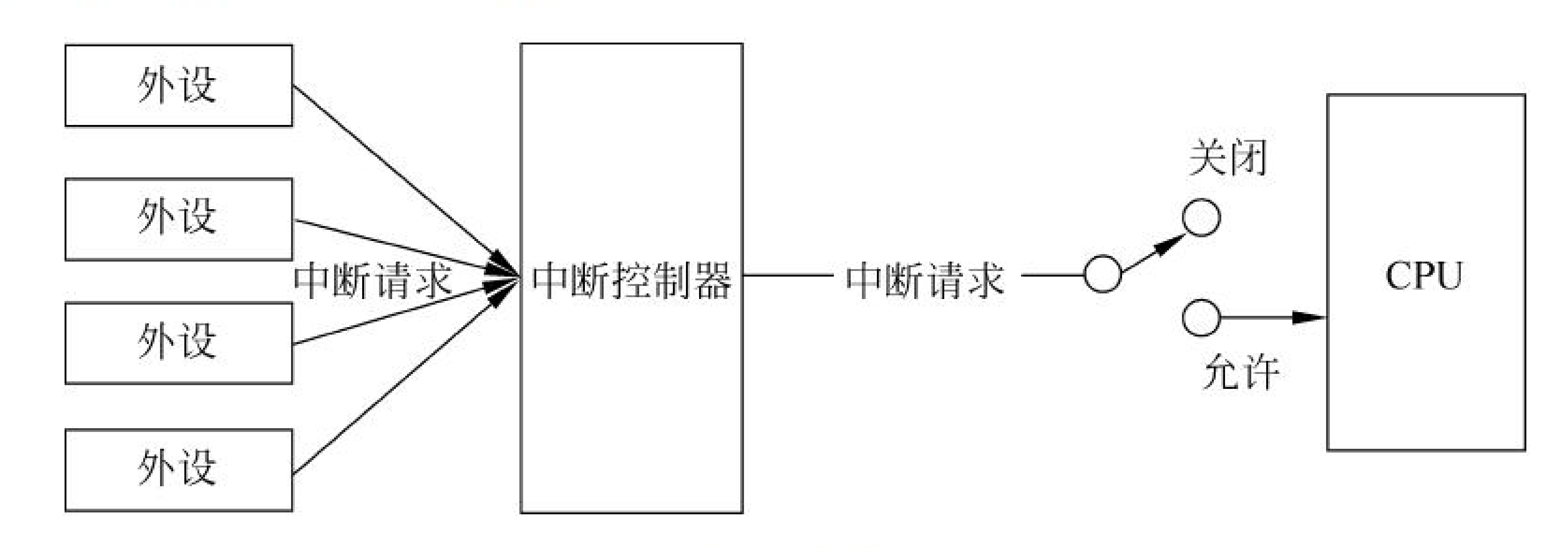


图 5.3　中断控制器运行过程

若关闭全部中断，CPU 将不再处理所有中断请求，但中断控制器会将这些中断请求保存下来，并在 CPU 再次打开中断后立即产生中断请求。CPU 有如下两种处理中断的模式。

(1) 所有的中断向量映射到一个共用的中断服务程序；

(2) 每个中断向量映射到各自的中断服务程序。

5.3 中断服务程序

μC/OS-Ⅲ的中断服务程序需要使用汇编语言编写，而且不同的微处理器会有不同的处理指令，所以这里只给出中断服务程序的示意性代码。支持 C 语言内嵌汇编的处理器也可以将汇编语言写在 C 语言中。典型的 μC/OS-Ⅲ中断服务程序示意性代码如下所示。

```
FirISR:
    Disable all interrupts;                    //关闭所有中断
    Save the CPU registers;                    //保存当前任务 CPU 的寄存器内容
    OSIntNestingCtr++;                         //中断嵌套层数加 1
    If(OSIntNestingCtr == 1){                  //中断嵌套层数为 1
        //保存被中断任务的堆栈指针到 OS_TCB 结构体中
      OSTCBCurPtr->StkPtr = Current task's CPU stack pointer register value;
    }
```

```
        Clear interrupting device;                //清除中断请求
        Re-enable interrupts(optional);           //允许中断嵌套
        Call user ISR;                            //调用用户中断服务程序
        OSIntExit();                              //中断服务程序退出,通知内核
        Restore the CPU registers;                //恢复 CPU 寄存器内容
        Return from interrupt;                    //从中断返回
```

很多情况下,中断需要做的处理非常简单,无须在任务代码中进行,这种情况下的中断服务程序代码如下:

```
SecISR:
    //保存本中断服务程序需要的寄存器
    Save enough registers as needed by the ISR;
    Clear interrupting device;                //清除设备中断
    DO NOT re-enable interrupts;              //不允许中断嵌套
    Call user ISR;                            //调用用户中断服务程序
    Restore the saved CPU registers;          //恢复寄存器内容
    Return from interrupt;                    //从中断返回
```

5.4 直接发布和延迟发布

μC/OS-Ⅲ的中断消息发布模式存在直接发布和延迟发布两种方式。用户可以通过设置宏 OS_CFG_ISR_POST_DEFERRED_EN 的值来配置发布方式,设置宏 OS_CFG_ISR_POST_DEFERRED_EN 的值为 1,表示使用延迟发布;设置为 0,则使用直接发布。

5.4.1 直接发布

μC/OS-Ⅱ就使用了直接发布模式,μC/OS-Ⅲ也保留了这个模式。图 5.4 为任务采用直接发布模式的示意图。

外设产生中断请求(1),μC/OS-Ⅲ响应中断,跳转到中断服务程序(2),中断服务程序可能释放一些信号量使等待的相关程序进入就绪状态,如果就绪任务的优先级比当前任务低或者相等,那么被中断的任务继续执行(3);如果就绪任务的优先级高于当前任务,那么新的更高优先级的任务被首先执行(4)。确定待执行任务的过程需要 μC/OS-Ⅲ系统关闭中断以保护临界区代码(5)。

当程序调用 μC/OS-Ⅲ的系统功能函数时,这些功能函数很可能会关中断。当 OS_CFG_ISR_POST_DEFERRED_EN 设置为 0 时,μC/OS-Ⅲ会对临界段代码采用关闭中断的保护措施,这样就会延长中断的响应时间。虽然 μC/OS-Ⅲ已经采取了所有可能的措施来缩短中断关闭时间,但仍然有一些复杂的 μC/OS-Ⅲ功能会使得中断关闭需要相对较长

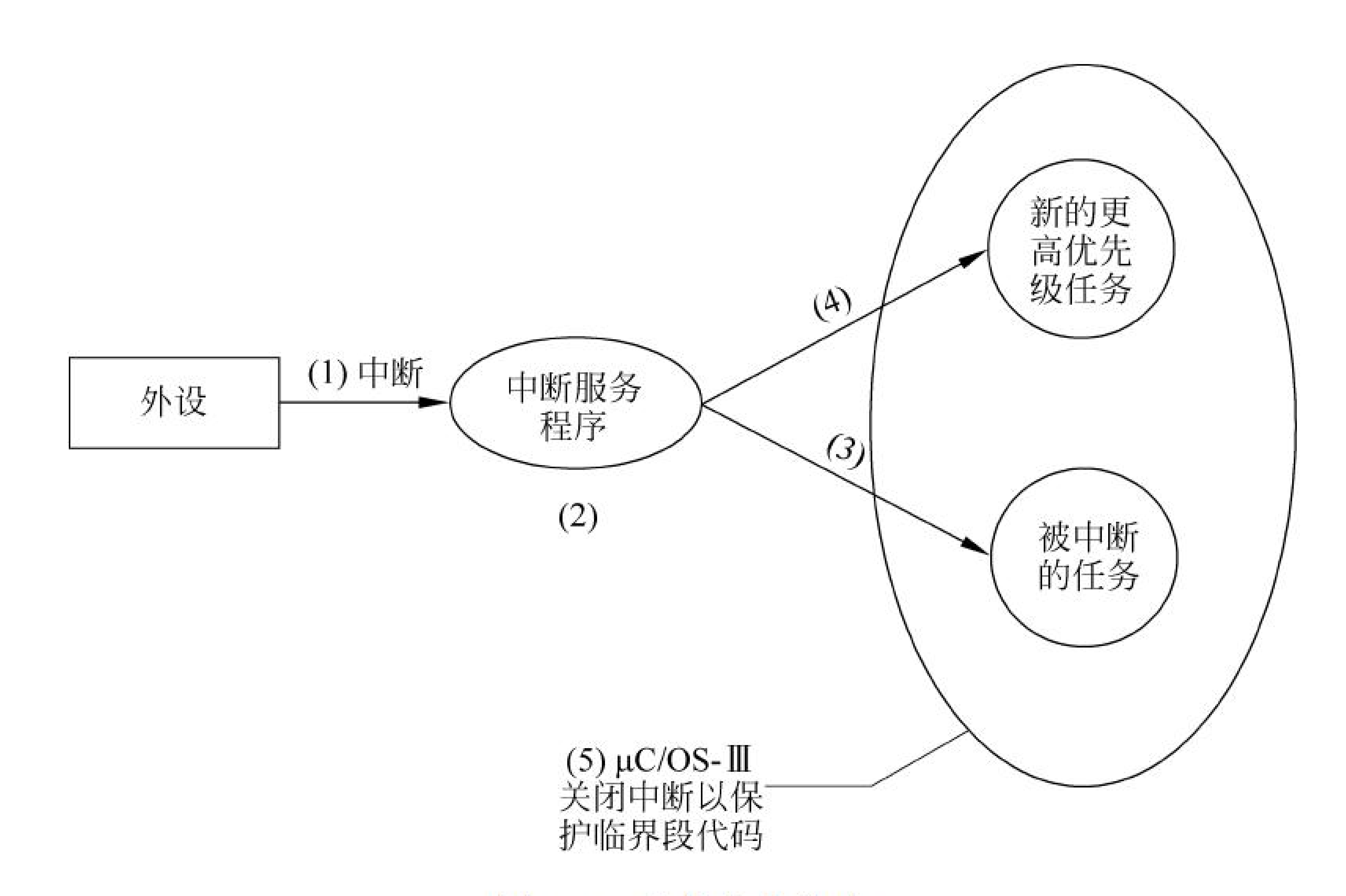


图 5.4 直接发布模式

的时间。是否使用直接发布模式的主要取决于中断关闭时间，通过系统提供的统计功能可得到 μC/OS-Ⅲ的中断关闭时间。

下面给出中断延迟时间(interrupt latency)、中断响应时间(interrupt response)、中断恢复时间(interrupt recovery)和任务延迟时间的计算方法。

(1) 中断延迟时间＝最大中断关闭时间；

(2) 中断响应时间＝中断延迟时间＋中断向量映射时间＋中断预处理时间；

(3) 中断恢复时间＝产生中断的设备处理时间＋给任务发布消息或信号时间＋OSIntExit()＋OSIntCtxSW()；

(4) 任务延迟时间＝中断响应时间＋中断恢复时间＋任务调度锁定时间。

5.4.2 延迟发布

在延迟模式下(OS_CFG_POST_DEFERRED_EN 设置为 1)，μC/OS-Ⅲ不是通过中断，而是通过给任务调度器上锁的方式来保护临界段代码，既保证了中断的响应和处理，也有效地保护了临界段代码。在延迟发布模式下，一般不存在关闭中断的情况。延迟发布模式的机制要比直接发布模式复杂，如图 5.5 所示。

外设产生中断请求(1)，μC/OS-Ⅲ响应中断，跳转到中断服务程序(2)，中断服务程序中可能使用相关的提交函数，例如 OSSemPost、OSQPost 等，这些函数可能会让某些等待任务进入就绪状态。延迟处理模式会将这些提交函数保存到中断队列中(3)，同时，μC/OS-Ⅲ还创建了中断处理任务 OS_IntQTask，该任务拥有最高的任务优先级 0，退出中断后，中断处理任务会被首先执行(4)。中断处理任务将中断队列中的任务处理完成后将自身挂起，并重新启动任务调度器来决定将要运行的任务，如果就绪的任务优先级都比被中断任务的优先级低或者相等，则被中断任务继续执行(5)；如果有新的更高优先级的任务就绪，则新的更

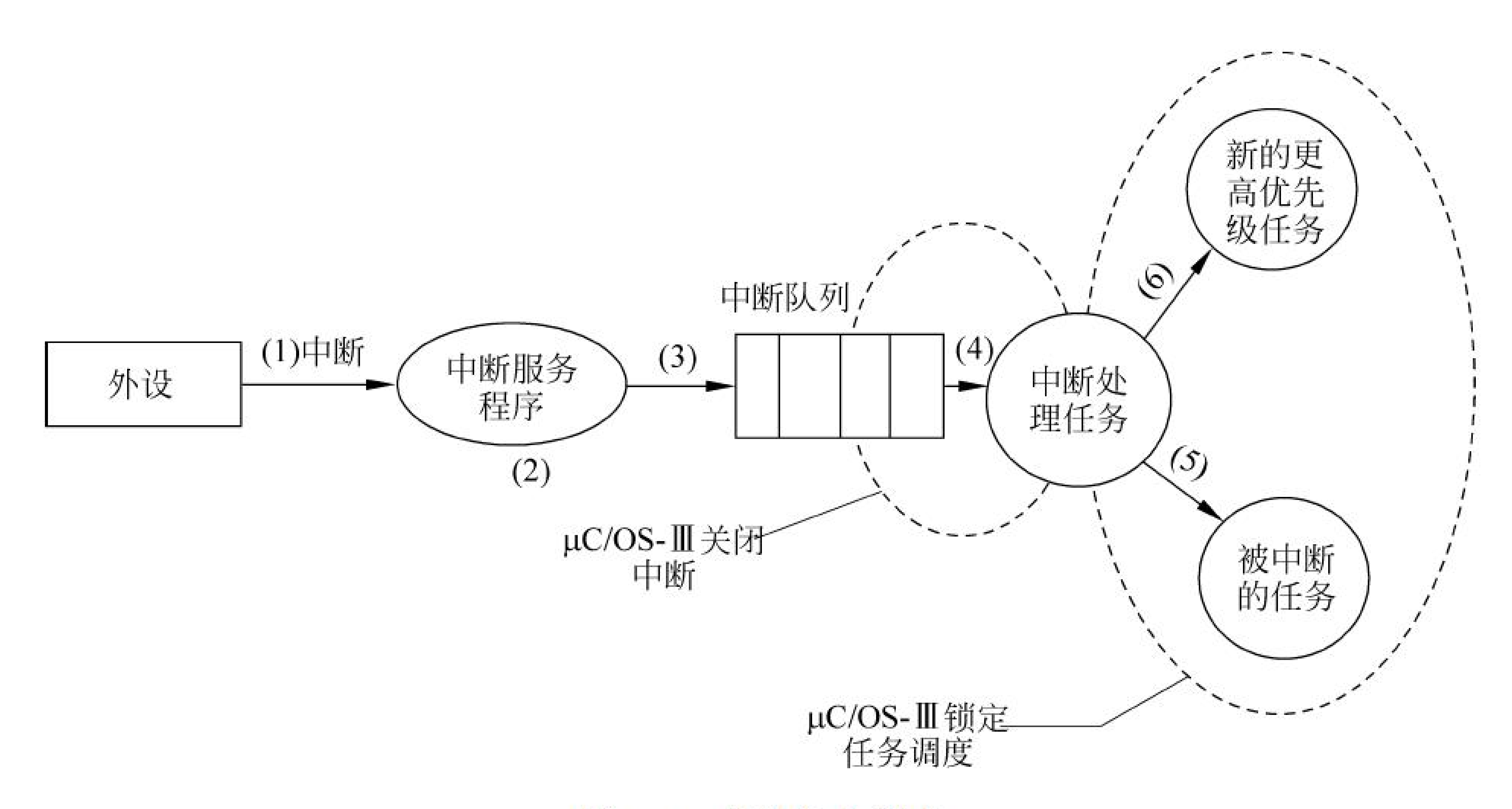


图 5.5　延迟发布模式

高优先级的任务被执行(6)。

所有额外增加的操作都是为了避免使用关中断的方法来保护临界段代码。这些额外增加的操作仅包括将发布调用及其参数复制到中断队列中,从中断队列提取发布调用和相关参数以及一次额外的任务切换。

与直接发布模式相似,下面给出延迟发布模式下的中断延迟时间、中断响应时间、中断恢复时间和任务延迟时间的计算方法。

(1) 中断延迟时间＝最大中断关闭时间;

(2) 中断响应时间＝中断延迟时间＋中断向量映射时间＋中断预处理时间;

(3) 中断恢复时间＝产生中断的设备处理时间＋向中断队列写入发布函数调用和相关参数的时间＋OSIntExit()＋OSIntCtxSW()(切换到中断队列处理任务的时间);

(4) 任务延迟时间＝中断响应时间＋中断恢复时间＋重新调用发布函数的时间＋任务切换时间＋任务调度锁定时间。

主要的差异在于延迟发布模式缩短了关闭中断的时间,因而缩短了中断延迟、中断响应和中断恢复的时间。但由于使用给任务调度器上锁的方式保护临界段代码,反而使得任务延迟时间变长了。

5.4.3　延迟提交信息记录块

```
struct  os_int_q {
    OS_OBJ_TYPE  Type;              //用于记录提交的内核对象类型
    //指向下一个延迟提交信息块 os_int_q 的指针
    OS_INT_Q      * NextPtr;
    void          * ObjPtr;         //指向内核对象变量指针
    void          * MsgPtr;         //如果发布的是消息,指向发布消息的指针
```

```
        //如果发布的是消息,代表发布消息的字节大小
        OS_MSG_SIZE  MsgSize;
        //如果发布的是事件标志,这个变量代表要设置的位
        OS_FLAGS      Flags;
        OS_OPT        Opt;            //内核选项
        CPU_TS        TS;             //时间戳
};
```

5.5　中断管理内部函数

5.5.1　中断进入函数

```
void  OSIntEnter(void)
```

功能描述：

该函数通常在保存了寄存器内容，保护了任务现场后被调用，是进入用户 dinginess 的中断服务程序。

注意：

(1) 调用该函数时，必须关闭中断；

(2) OSIntEnter()和 OSIntExit()必须成对出现；

(3) 中断嵌套层数必须小于 250。

```
void  OSIntEnter(void)
{
    //判断是否为运行状态
    if (OSRunning != OS_STATE_OS_RUNNING) {
        return;
    }
    //嵌套层数大于 250
    if (OSIntNestingCtr >= (OS_NESTING_CTR)250u) {
        return;
    }
    //嵌套层数加 1
    OSIntNestingCtr++;
}
```

5.5.2　中断退出函数

```
void  OSIntExit(void)
```

功能描述：

该函数被用来通知μC/OS-Ⅲ内核中断服务程序已经完成，当中断服务程序的最后一层嵌套完成时，μC/OS-Ⅲ将调用调度器使优先级最高的就绪任务准备运行。

注意：

（1）OSIntEnter()和OSIntExit()必须成对出现，即使用OSIntEnter()进入中断，必须使用OSIntExit()退出中断；

（2）当调度器被锁定时，任务的重新调度被禁止。

```
void  OSIntExit(void)
{
    CPU_SR_ALLOC();
    //系统不处于运行状态
    if (OSRunning != OS_STATE_OS_RUNNING) {
        return;
    }
    CPU_INT_DIS();
    //中断嵌套计数等于 0
    if (OSIntNestingCtr == (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }
    //中断嵌套计数减 1
    OSIntNestingCtr--;
    //中断嵌套计数大于 0
    if (OSIntNestingCtr > (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }
    //调度器锁嵌套大于 0
    if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
        CPU_INT_EN();
        return;
    }
    //此时没有中断嵌套并且调度器没有上锁
    //获取就绪任务的最高优先级
    OSPrioHighRdy = OS_PrioGetHighest();
    //获取就绪最高优先级任务的指针
    OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
    //最高优先级就绪任务是当前任务
    if (OSTCBHighRdyPtr == OSTCBCurPtr) {
        CPU_INT_EN();
        return;
    }
```

```
#if OS_CFG_TASK_PROFILE_EN > 0u
    //最高优先级就绪任务的上下文切换计数加 1
    OSTCBHighRdyPtr->CtxSwCtr++;
#endif
    //当前任务上下文切换计数加 1
    OSTaskCtxSwCtr++;
#if defined(OS_CFG_TLS_TBL_SIZE) && (OS_CFG_TLS_TBL_SIZE > 0u)
    OS_TLS_TaskSw();
#endif
    //执行中断级任务切换
    OSIntCtxSw();
    CPU_INT_EN();
}
```

5.5.3 中断级任务切换函数

```
void  OSIntCtxSw (void)
```

功能描述：

该函数被 OSIntExit() 函数调用来在中断服务程序中执行任务上下文切换。OSIntCtxSW()首先调用 OSTaskSwHook()，用户可以在该函数中添加一些其他在上下文切换期间执行的操作，再把当前任务指针设置为最高优先级任务的指针，当前任务优先级设置为最高优先级就绪任务的优先级。

```
void  OSIntCtxSw(void)
{
    //任务上下文切换钩子函数
    OSTaskSwHook();
    //当前任务指针设置为最高优先级任务指针
    OSTCBCurPtr = OSTCBHighRdyPtr;
    //当前优先级等于最高优先级
    OSPrioCur = OSPrioHighRdy;
}
```

5.5.4 临界区进入和退出宏

功能描述：OS_CRITICAL_ENTER()宏被调用标识程序进入临界区，可以访问和修改公共变量。调用 OS_CRITICAL_ENTER_CPU_EXIT()宏同样可以进入临界区，但是中断被重新使能。OS_CRITICAL_EXIT()宏退出临界区，退出临界区时进行任务调度，OS_CRITICAL_EXIT_NO_SCHED()宏同样是退出临界区，但是不进行任务调度。

```
#define  OS_CRITICAL_ENTER()
 do {
      CPU_CRITICAL_ENTER();                    //CPU 进入临界区
      OSSchedLockNestingCtr++;                 //调度锁嵌套层数加 1
      if (OSSchedLockNestingCtr  == 1u) {      //调度锁嵌套层数等于 1
      //启动调度锁锁定时间测量
        OS_SCHED_LOCK_TIME_MEAS_START();
      }
      CPU_CRITICAL_EXIT();                     //CPU 退出临界区
} while (0)

#define  OS_CRITICAL_ENTER_CPU_EXIT()
 do {
      OSSchedLockNestingCtr++;                 //调度锁嵌套层数加 1
      if (OSSchedLockNestingCtr  == 1u) {      //调度锁嵌套层数等于 1
      //启动调度锁锁定时间测量
          OS_SCHED_LOCK_TIME_MEAS_START();
      }
      CPU_CRITICAL_EXIT();
} while (0)

#define  OS_CRITICAL_EXIT()
 do {
      CPU_CRITICAL_ENTER();                    //CPU 进入临界区
      OSSchedLockNestingCtr--;                 //调度锁嵌套层数减 1
      //调度锁嵌套层数等于 0
      if (OSSchedLockNestingCtr  == (OS_NESTING_CTR)0) {
          //调度锁锁定时间测量停止
          OS_SCHED_LOCK_TIME_MEAS_STOP();
          //延迟中断处理队列中的对象数目大于 0
          if (OSIntQNbrEntries > (OS_OBJ_QTY)0) {
              CPU_CRITICAL_EXIT();             //CPU 退出临界区
              OS_Sched0();                     //进行任务调度
          } else {
              CPU_CRITICAL_EXIT();             //CPU 退出临界区
          }
      } else {
          CPU_CRITICAL_EXIT();
      }
} while (0)

#define  OS_CRITICAL_EXIT_NO_SCHED()
 do {
      CPU_CRITICAL_ENTER();                    //CPU 进入临界区
      OSSchedLockNestingCtr--;                 //调度锁嵌套层数减 1
      //调度锁嵌套层数等于 0
```

```
        if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {
            //调度锁锁定时间测量停止
            OS_SCHED_LOCK_TIME_MEAS_STOP();
        }
        CPU_CRITICAL_EXIT();    //CPU 退出临界区
} while (0)
```

5.5.5 中断延迟队列初始化函数

```
void OS_IntQTaskInit(OS_ERR *p_err)
```

功能描述：

该函数被系统初始化函数 OSInit()调用来初始化中断服务程序队列。

参数描述：

p_err：指向可能出现的错误代码的指针。

错误类型如下：

(1) OS_ERR_INT_Q：在 OS_CFG.C 中没有提供一个中断队列；

(2) OS_ERR_INT_Q_SIZE：中断服务程序队列空间不足；

(3) OS_ERR_STK_INVALID：指向 ISR 任务栈的指针为空；

(4) OS_ERR_STK_SIZE_INVALID：指定任务栈大小小于最小要求。

```
void  OS_IntQTaskInit(OS_ERR   *p_err)
{
    //指向延迟提交信息记录块
    OS_INT_Q        *p_int_q;
    OS_INT_Q        *p_int_q_next;
    OS_OBJ_QTY       i;
    //清空中断服务程序队列的溢出计数值
    OSIntQOvfCtr = (OS_QTY)0u;
    //延迟提交信息队列的基地址指针为空
    if (OSCfg_IntQBasePtr == (OS_INT_Q *)0) {
        *p_err = OS_ERR_INT_Q;
        return;
    }
    //延迟提交信息队列的成员数小于 2
    if (OSCfg_IntQSize < (OS_OBJ_QTY)2u) {
        *p_err = OS_ERR_INT_Q_SIZE;
        return;
    }
    //延迟提交信息队列中任务的最大运行时间
    OSIntQTaskTimeMax = (CPU_TS)0;
```

```
//p_int_q 指向延迟提交队列的基地址
p_int_q = OSCfg_IntQBasePtr;
p_int_q_next = p_int_q;
//p_int_q_next 指向下一个延迟提交记录块
p_int_q_next++;
//初始化一个循环延迟提交信息记录块链表
//初始化延迟提交队列中的各个信息记录块
for (i = 0u; i < OSCfg_IntQSize; i++) {
    p_int_q->Type = OS_OBJ_TYPE_NONE;
    p_int_q->ObjPtr = (void *)0;
    p_int_q->MsgPtr = (void *)0;
    p_int_q->MsgSize = (OS_MSG_SIZE)0u;
    p_int_q->Flags = (OS_FLAGS )0u;
    p_int_q->Opt = (OS_OPT )0u;
    p_int_q->NextPtr = p_int_q_next;
    p_int_q++;
    p_int_q_next++;
}
//p_int_q 指向最后一个延迟提交信息记录块
p_int_q--;
//p_int_q_next 指向队列的基地址
p_int_q_next = OSCfg_IntQBasePtr;
//p_int_q 的后继指针指向 p_int_q_next,形成循环链表
p_int_q->NextPtr = p_int_q_next;
//队列的入口地址指向 OSIntQInPtr = p_int_q_next; p_int_q_next
//队列的出口地址指向 p_int_q_next
OSIntQOutPtr = p_int_q_next;
//队列中包含的记录块个数为 0
OSIntQNbrEntries = (OS_OBJ_QTY)0u;
//队列中包含的最大记录块个数为 0
OSIntQNbrEntriesMax = (OS_OBJ_QTY)0u;
//创建中断服务程序队列处理任务
//任务栈基地址等于空
if (OSCfg_IntQTaskStkBasePtr == (CPU_STK *)0) {
    *p_err = OS_ERR_INT_Q_STK_INVALID;
    return;
}
//任务栈大小小于任务栈的最小空间
if (OSCfg_IntQTaskStkSize < OSCfg_StkSizeMin) {
    *p_err = OS_ERR_INT_Q_STK_SIZE_INVALID;
    return;
}
//创建中断服务程序队列处理任务
OSTaskCreate(
    (OS_TCB *)&OSIntQTaskTCB,
    (CPU_CHAR *)((void *)"uC/OS-Ⅲ ISR Queue Task"),
```

```
        (OS_TASK_PTR)OS_IntQTask,       //任务名称
        (void *)0,
        (OS_PRIO)0u,                    //该任务的优先级为 0,是最高优先级
        (CPU_STK *)OSCfg_IntQTaskStkBasePtr,
        (CPU_STK_SIZE)OSCfg_IntQTaskStkLimit,
        (CPU_STK_SIZE)OSCfg_IntQTaskStkSize,
        (OS_MSG_QTY)0u,
        (OS_TICK)0u,
        (void *)0,
        (OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
        (OS_ERR *)p_err);
}
```

5.5.6 中断延迟队列提交函数

```
void  OS_IntQPost(OS_OBJ_TYPE   type,
                  void          *p_obj,
                  void          *p_void,
                  OS_MSG_SIZE   msg_size,
                  OS_FLAGS      flags,
                  OS_OPT        opt,
                  CPU_TS        ts,
                  OS_ERR        *p_err)
```

参数说明：

（1）type：传递的内核对象类型。

具体取值：

① OS_OBJ_TYPE_SEM；

② OS_OBJ_TYPE_Q；

③ OS_OBJ_TYPE_FLAG；

④ OS_OBJ_TYPE_TASK_MSG；

⑤ OS_OBJ_TYPE_TASK_SIGNAL。

（2）p_obj：指向要传递的内核对象，这可能是一个信号量、一个信息队列或者是一个任务控制时钟。

（3）p_void：指向要传递的消息。

（4）msg_size：要传递的消息大小。

（5）flags：表示传递事件标志组。

（6）ts：传递的时间戳。

（7）opt：传递的相应选项。

具体取值：

① OSFlagPost()；

② OSSemPost()；

③ OSQPost()；

④ OSTaskQPost()。

(8) p_err：表示传递过程中可能出现的错误。

具体取值：

① OS_ERR_NONE：没有错误；

② OS_ERR_INT_Q_FULL：中断服务程序队列已满。

功能描述：

该函数将要发送的内容传递到中断队列中，帮助中断延迟提交处理。

```
void  OS_IntQPost(OS_OBJ_TYPE   type,
                  void          *p_obj,
                  void          *p_void,
                  OS_MSG_SIZE   msg_size,
                  OS_FLAGS      flags,
                  OS_OPT        opt,
                  CPU_TS        ts,
                  OS_ERR        *p_err)
{
    CPU_SR_ALLOC();
    CPU_CRITICAL_ENTER();                           //进入临界区
    //确保中断延迟提交内核对象的数量没有超过最大值
    if (OSIntQNbrEntries < OSCfg_IntQSize){
        OSIntQNbrEntries++;                         //中断延迟提交内核对象的数量加 1
        //更新延迟提交内核对象的最大数目
        if (OSIntQNbrEntriesMax < OSIntQNbrEntries) {
            OSIntQNbrEntriesMax = OSIntQNbrEntries;
        }
  //将传递的内核对象参数保存到延迟提交内核对象队列的入口对象 OSIntQInPtr 中
        OSIntQInPtr->Type = type;                   //保存传递的内核对象类型
        OSIntQInPtr->ObjPtr = p_obj;                //保存传递的内核对象的指针
        //保存传递的指向消息队列的指针
        OSIntQInPtr->MsgPtr = p_void;
        //保存传递的消息队列的大小
        OSIntQInPtr->MsgSize = msg_size;
        OSIntQInPtr->Flags = flags;                 //如果传递的是 Flag,则保存
        OSIntQInPtr->Opt = opt;                     //保存传递的选项
        OSIntQInPtr->TS = ts;                       //保存时间戳
        //延迟提交内核对象队列入口指针 OSIntQInPtr 后移
        OSIntQInPtr = OSIntQInPtr->NextPtr;
        OSRdyList[0].NbrEntries = (OS_OBJ_QTY)1;
```

```
        OSRdyList[0].HeadPtr = &OSIntQTaskTCB;
        OSRdyList[0].TailPtr = &OSIntQTaskTCB;
        //在优先级列表中添加任务优先级 0,即 OS_IntQTask 任务
        OS_PrioInsert(0u);
        if (OSPrioCur != 0) {              //OS_IntQTask 任务没有运行
            OSPrioSaved = OSPrioCur;       //保存当前任务的优先级
        }
        *p_err = OS_ERR_NONE;
    } else {
        OSIntQOvfCtr++;                    //增加延迟提交内核对象队列溢出的次数
        *p_err = OS_ERR_INT_Q_FULL;
    }
    CPU_CRITICAL_EXIT();
}
```

5.5.7 中断延迟队列真正提交函数

```
void  OS_IntQRePost(void)
```

功能描述：

该函数实现了中断延迟队列任务的真正提交，被中断队列管理任务调用来提交任务。

```
void  OS_IntQRePost(void)
{
#if OS_CFG_TMR_EN > 0u                     //是否开启了软件定时器功能
    CPU_TS  ts;
#endif
    OS_ERR  err;
    //延迟提交队列出口指针指向的记录块类型
    switch (OSIntQOutPtr->Type) {
        case OS_OBJ_TYPE_FLAG:             //Flag 事件标志类型
#if OS_CFG_FLAG_EN > 0u
        //提交事件标志组
        (void)OS_FlagPost((OS_FLAG_GRP *) OSIntQOutPtr->ObjPtr,
                          (OS_FLAGS)OSIntQOutPtr->Flags,
                          (OS_OPT) OSIntQOutPtr->Opt,
                          (CPU_TS) OSIntQOutPtr->TS,
                          (OS_ERR *)&err);
#endif
            break;
        case OS_OBJ_TYPE_Q:                //消息类型
#if OS_CFG_Q_EN > 0u
    //提交消息
            OS_QPost((OS_Q *) OSIntQOutPtr->ObjPtr,
```

```
                        (void * ) OSIntQOutPtr -> MsgPtr,
                        (OS_MSG_SIZE) OSIntQOutPtr -> MsgSize,
                        (OS_OPT) OSIntQOutPtr -> Opt,
                        (CPU_TS) OSIntQOutPtr -> TS,
                        (OS_ERR * )&err);
#endif
                break;

            case OS_OBJ_TYPE_SEM:                //信号量类型
#if OS_CFG_SEM_EN > 0u
                //提交信号量
                (void)OS_SemPost((OS_SEM  * ) OSIntQOutPtr -> ObjPtr,
                     (OS_OPT) OSIntQOutPtr -> Opt,
    (CPU_TS) OSIntQOutPtr -> TS,
(OS_ERR * )&err);
#endif
                break;

            case OS_OBJ_TYPE_TASK_MSG:           //任务消息类型
#if OS_CFG_TASK_Q_EN > 0u
                //提交任务消息
                OS_TaskQPost((OS_TCB * ) OSIntQOutPtr -> ObjPtr,
                (void * ) OSIntQOutPtr -> MsgPtr,
                (OS_MSG_SIZE) OSIntQOutPtr -> MsgSize,
                (OS_OPT) OSIntQOutPtr -> Opt,
                (CPU_TS) OSIntQOutPtr -> TS,
                (OS_ERR * )&err);
#endif
break;

            case OS_OBJ_TYPE_TASK_RESUME:        //任务唤醒类型
#if OS_CFG_TASK_SUSPEND_EN > 0u
                //唤醒任务
                (void)OS_TaskResume((OS_TCB  * ) OSIntQOutPtr -> ObjPtr,
                (OS_ERR  * )&err);
#endif
                break;

            case OS_OBJ_TYPE_TASK_SIGNAL:        //信号类型
    //发送信号
                (void)OS_TaskSemPost((OS_TCB  * ) OSIntQOutPtr -> ObjPtr,
(OS_OPT) OSIntQOutPtr -> Opt,
(CPU_TS) OSIntQOutPtr -> TS,
(OS_ERR * )&err);
                break;
```

```
        case OS_OBJ_TYPE_TASK_SUSPEND:    //任务挂起类型
#if OS_CFG_TASK_SUSPEND_EN > 0u
        //挂起任务
            (void)OS_TaskSuspend((OS_TCB *) OSIntQOutPtr->ObjPtr,
    (OS_ERR *)&err);
#endif
break;

        case OS_OBJ_TYPE_TICK:             //任务节拍类型
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u
            //RR 调度就绪链表中的最高优先级任务
            OS_SchedRoundRobin(&OSRdyList[OSPrioSaved]);
#endif
    //发送信号给 OSTickTask 任务
            (void)OS_TaskSemPost((OS_TCB *)&OSTickTaskTCB,
(OS_OPT) OS_OPT_POST_NONE,
(CPU_TS) OSIntQOutPtr->TS,
(OS_ERR *)&err);
#if OS_CFG_TMR_EN > 0u

            OSTmrUpdateCtr--;
            if (OSTmrUpdateCtr == (OS_CTR)0u) {
                OSTmrUpdateCtr = OSTmrUpdateCnt;
                ts             = OS_TS_GET();
//发送信号给定时器任务
                (void)OS_TaskSemPost((OS_TCB *)&OSTmrTaskTCB,
                                     (OS_OPT  ) OS_OPT_POST_NONE,
                                     (CPU_TS  ) ts,
                                     (OS_ERR *)&err);
            }
#endif
            break;

        default:
            break;
    }
}
```

5.5.8　中断队列管理任务

```
void  OS_IntQTask(void  *p_arg)
```

参数描述：

p_arg：指向可选的参数。

功能描述：

该函数用于管理延迟中断队列中的任务，将任务要发送的请求通过 OS_IntQRePost()函数发送，并将任务从延迟中断队列中移出，同时测量了任务的执行时间。

```
void  OS_IntQTask(void  *p_arg)
{
    CPU_BOOLEAN  done;                          //完成标识
    CPU_TS       ts_start;                      //开始时间
    CPU_TS       ts_end;                        //完成时间
    CPU_SR_ALLOC();
    (void)&p_arg;                               //不使用 p_arg,主要是防止编译错误
    while (DEF_ON) {
        done = DEF_FALSE;
        while (done == DEF_FALSE) {             //没有完成
            CPU_CRITICAL_ENTER();               //CPU 进入临界区
            //延迟中断队列中对象个数等于 0
            if (OSIntQNbrEntries == (OS_OBJ_QTY)0u) {
            //将管理任务从就绪队列中移出
                OSRdyList[0].NbrEntries = (OS_OBJ_QTY)0u;
                OSRdyList[0].HeadPtr = (OS_TCB   *)0;
                OSRdyList[0].TailPtr = (OS_TCB   *)0;
                OS_PrioRemove(0u);              //将管理任务从优先级表中移出
                CPU_CRITICAL_EXIT();            //CPU 退出临界区
                OSSched();                      //执行调度程序
                done = DEF_TRUE;                //队列中没有对象,完成
            } else {
                CPU_CRITICAL_EXIT();
                ts_start = OS_TS_GET();         //获取开始时间
                OS_IntQRePost();                //中断延迟队列发送请求
                ts_end = OS_TS_GET() - ts_start;  //测量任务执行时间
                if (OSIntQTaskTimeMax < ts_end) {
                    OSIntQTaskTimeMax = ts_end;
                }
                CPU_CRITICAL_ENTER();           //CPU 进入临界区
                //出口指针指向出口队列中的下一个对象
                OSIntQOutPtr = OSIntQOutPtr->NextPtr;
                //延迟中断队列中对象个数减 1
                OSIntQNbrEntries--;
                //退出临界区
                CPU_CRITICAL_EXIT();
            }
        }
    }
}
```

习题

1. 操作系统的中断机制是什么？为什么需要中断？

2. 中断机制与轮询处理机制的区别是什么？中断机制和轮询机制分别适用于哪些情况？

3. 请详述 μC/OS-Ⅲ的中断处理流程。

4. 中断服务程序中要进行哪些必要的处理？

5. μC/OS-Ⅲ中的中断直接发布与延迟发布的区别是什么？如何看待延迟发布机制？

6. 延迟提交信息记录块 os_int_q 的内容是什么？

7. μC/OS-Ⅲ内核是如何实现中断进入和中断退出机制的？

8. 中断任务级切换主要包含哪些操作？

9. 临界区进入和退出是如何实现的？

10. 中断延迟队列是如何初始化的？延迟中断的提交采用了什么方法？

第 6 章



时 钟 管 理

6.1 总体描述

时钟管理是操作系统中非常重要的部分。操作系统中既包含事件驱动的任务，例如当按下一个按钮时，需要按键中断处理程序去处理；同时也包含时间驱动的任务，例如在某一时间执行特定的任务。时间驱动的任务是基于时钟节拍的，在指定的时间，任务根据给定的节拍数挂起或者执行一段时间。任何一个操作系统都需要时钟节拍，μC/OS-Ⅲ也不例外。一个时钟节拍是指两个时钟中断发生之间的时间，每个时钟节拍发生后，内核会产生一次时钟中断。内核可以通过时钟节拍实现时钟管理，对时钟任务进行管理主要通过延时任务、定时器、时间片调度任务等来实现。

其实，内核本身并没有时间的概念，因为时钟节拍是由硬件产生的，内核通过计算时钟节拍从而得到系统时间。时钟节拍应该被赋予一个合理的值，过快的时钟节拍虽然可以更加准确地控制时间，但无疑也增加了系统的负担；过慢的时钟节拍会导致时间精度不够准确，使得一些任务不能够被及时调度。例如，假设存在任务 A 因为等待系统资源而被挂起，系统的时钟节拍为 1Hz，那么时间中断每 1s 执行一次。在上个时间中断执行完成 0.1s 后，任务 A 获得相关资源变为就绪态并且拥有最高的优先级，但是任务 A 不能被立即调度执行，需要等到下一次时钟中断发生。一般推荐时钟节拍的频率为 10～1000Hz。用户可以在 os_cfg_app.h 文件中修改 OS_CFG_TICK_RATE_HZ 宏来修改时钟节拍的频率。

```
#define  OS_CFG_TICK_RATE_HZ            1000u
```

μC/OS-Ⅲ是一个基于优先级的可抢占式硬实时内核，并且 μC/OS-Ⅲ上运行的任务一般是无限循环任务，为了阻止高优先级的任务一直独占 CPU，保证低优先级的任务也能够顺利执行，μC/OS-Ⅲ中除空闲任务外的所有任务都必须在合适的位置调用系统提供的延时函数或者任务调度函数，让当前的任务暂停运行一段时间后进行一个任务切换。在 μC/OS-Ⅲ中，任务可以调用时间延迟函数，例如：OSTimeDly()、OSTimeDlyHMSM()等函数，将自己挂起一段时间。这时任务会由运行态切换到等待态，插入到时基列表中，因此任务在延时

过程中将会放弃对 CPU 的使用权。当任务延时结束后,会进入到就绪态,等待内核的下一次调用。

时钟管理主要实现了对延时任务进行管理,通过计算是否到达预定延时值,并根据任务的状态,更新其延时剩余节拍数、任务状态标志等信息,确定继续延时还是离开延时进入其他任务状态。

时钟管理机制如图 6.1 所示。

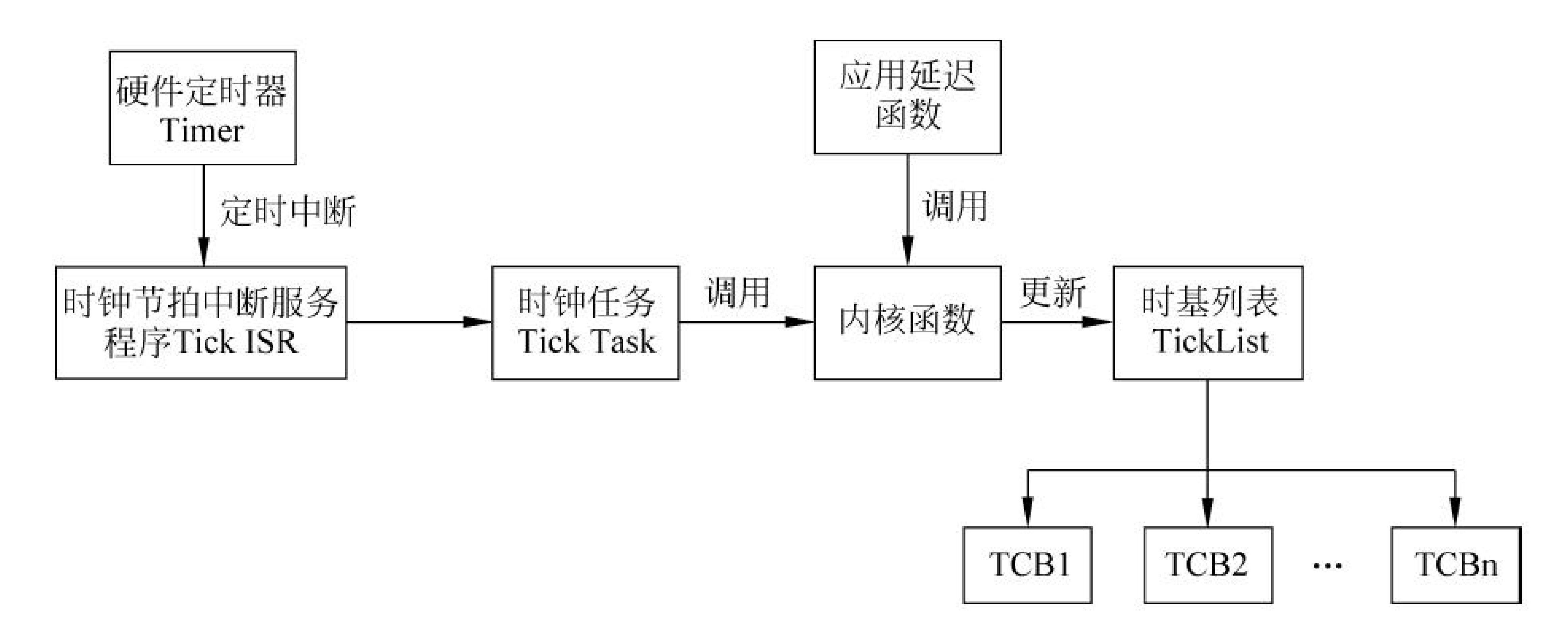


图 6.1 μC/OS-Ⅲ时钟管理机制

6.2 时钟机制分析

6.2.1 结构体 os_tick_spoke

结构体 os_tick_spoke 用于管理延时结束时刻具有相同属性的任务。

```
struct  os_tick_spoke
{
    OS_TCB   *FirstPtr;
    OS_OBJ_QTY  NbrEntries;
    OS_OBJ_QTY  NbrEntriesMax;
}
```

参数说明:

(1) FirstPtr:指向延时任务或指定超时时限的等待任务 TCB 的指针;

(2) NbrEntries:记录当前连接在链表上的任务数目;

(3) NbrEntriesMax:此 spoke 指向的链表上最大的任务数目。

图 6.2 描述了 spoke 的具体结构。

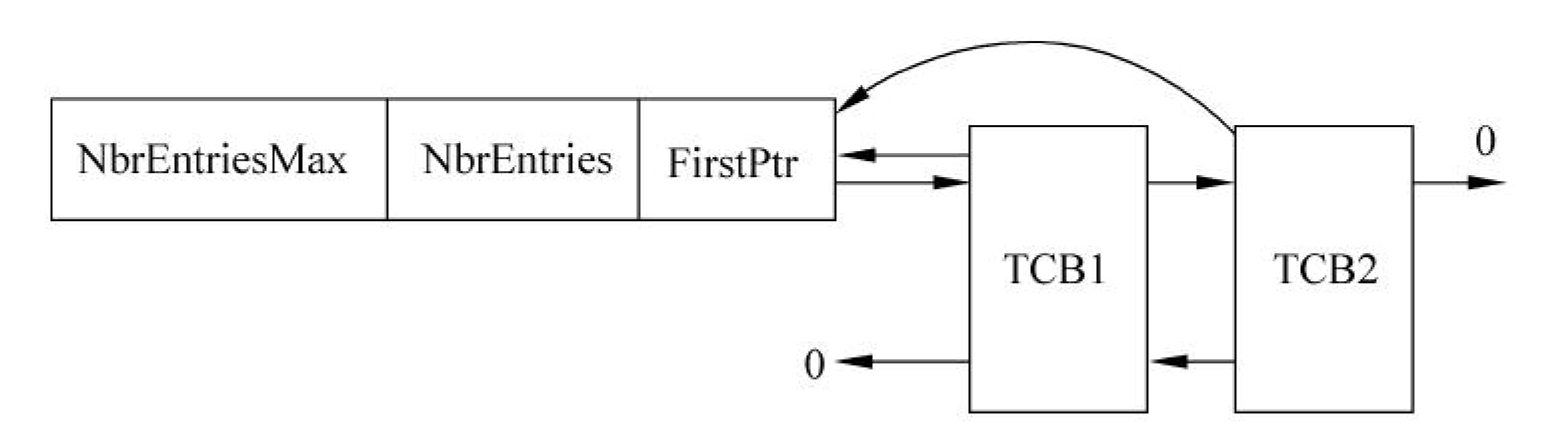


图 6.2 os_tick_spoke 结构

6.2.2 时钟任务管理

μC/OS-Ⅲ在运行过程中会创建时钟节拍任务 OS_TickTask(),负责处理延时任务和指定超时时限的等待任务。当时钟节拍用完后,系统会调用时钟节拍中断服务程序,并在其中调用 OSTimeTick()。针对时钟节拍,OSTimeTick()会通过 OSTaskSemPost()向时钟节拍任务 OS_TickTask()发送信号。通常时钟节拍任务 OS_TickTask()是处于挂起状态即等待 OSTimeTick 的信号状态,当得到信号后,时钟节拍任务会进入运行态,调用 OS_TickListUpdate()更新时基任务列表。

μC/OS-Ⅲ中使用哈希散列表结构管理所有正在延时的任务和指定了超时时限的等待任务。将所有延时任务记录在时基列表 TickList 中,由时钟节拍轮数组 OSCfg_TickWheel[](由 os_tick_spoke 构成的数组)和时钟节拍计数器 OSTickCtr 组成。每个 spoke 中存在一个数据成员 FirstPtr 指向一个双向链表,用于管理映射到 OSCfg_TickWheel 数组中同一条目内的延时任务。双向链表根据等待时间由长到短对任务进行排序,延时结束时间最早的任务放在链表前面。对时钟节拍任务进行扫描时,只需要定位到某个链表,并在链表上查找即可,不用扫描全部任务,大大减少了处理时间,如图 6.3 所示。

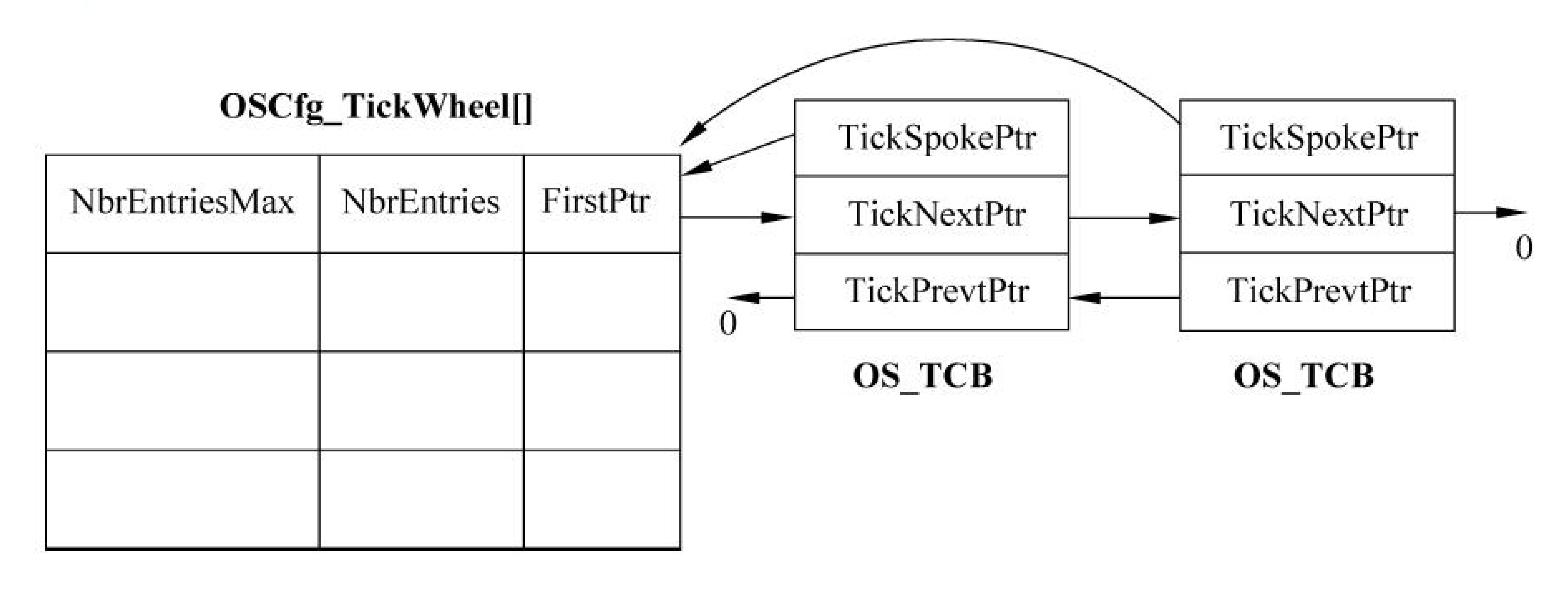


图 6.3 延时任务列表

6.2.3 延时任务 TCB

延时任务的 TCB 结构和普通任务的 TCB 并没有什么不同,它对和时钟节拍部分相关的参数进行了赋值。

```
struct os_tcb{
    OS_TCB      * TickNextPtr;
    OS_TCB      * TickPrevPtr;
    OS_TICK_SPOKE       * TickSpokePtr;
    OS_TICK     TickCtrPrev;
    OS_TICK     TickCtrMatch;
    OS_TICK     TickRemain;
}
```

参数说明：

(1) TickNextPtr：指向链表中下一个延时任务或指定超时时限的等待任务的指针；

(2) TickPrevPtr：指向链表中上一个延时任务或指定超时时限的等待任务的指针；

(3) TickSpokePtr：指向挂在此任务的 os_tick_spoke；

(4) TickCtrMatch：任务转向就绪态的匹配时间；

(5) TickRemain：到达匹配时间剩余的节拍数＝TickCtrMatch－OSTickCtr。

6.3　时钟管理内核函数

6.3.1　时钟节拍中断函数

```
void OSTimeTick(void)
```

功能描述：

系统根据硬件定时器产生的节拍中断，每次调用此函数向时钟节拍任务函数 OS_TickTask()发送信号，使得时钟节拍任务执行。此函数只能由时钟中断处理函数 OS_CPU_SysTickHandler()即 Tick ISR 调用。

```
void  OSTimeTick(void)
{
    OS_ERR  err;
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u             //开启了中断处理任务
    CPU_TS  ts;
#endif
    OSTimeTickHook();                           //调用用户定义的钩子函数
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u             //开启了中断处理任务延迟发布形式

    ts = OS_TS_GET();                           //获取系统时间戳
    //发送到中断处理函数队列中
    OS_IntQPost((OS_OBJ_TYPE) OS_OBJ_TYPE_TICK,
                (void * )&OSRdyList[OSPrioCur],
                (void * ) 0,
```

```
                      (OS_MSG_SIZE) 0u,
                      (OS_FLAGS) 0u,
                      (OS_OPT) 0u,
                      (CPU_TS) ts,
                      (OS_ERR * )&err);
#else
    (void)OSTaskSemPost((OS_TCB * )&OSTickTaskTCB,    //发送任务信号量
                        (OS_OPT) OS_OPT_POST_NONE,
                        (OS_ERR * )&err);
#if OS_CFG_SCHED_ROUND_ROBIN_EN > 0u                  //开启 RR 调度
    OS_SchedRoundRobin(&OSRdyList[OSPrioCur]);        //使用 RR 调度
#endif

#if OS_CFG_TMR_EN > 0u
    OSTmrUpdateCtr--;
    if (OSTmrUpdateCtr == (OS_CTR)0u) {
        OSTmrUpdateCtr = OSTmrUpdateCnt;
        OSTaskSemPost((OS_TCB * )&OSTmrTaskTCB,
                      (OS_OPT) OS_OPT_POST_NONE,
                      (OS_ERR * )&err);
    }
#endif
#endif
}
```

函数运行时首先会调用 OSTimeTickHook()函数,运行用户在给定的钩子函数中写入的代码。如果 OS_CFG_ISR_POST_DEFERRED_EN > 0u,说明中断将采取延迟发布的形式,OSTimeTick()将会发送一个请求到中断处理队列中。如果没有开启中断延迟处理,则采用信号量的方法发送。

6.3.2 时钟节拍任务

```
void  OS_TickTask(void  * p_arg)
```

参数说明:

p_arg:传递给时钟节拍任务的参数。

功能描述:

该函数被 μC/OS-Ⅲ系统内部函数调用来产生时钟中断。

```
void  OS_TickTask(void  * p_arg)
{
    OS_ERR   err;
    CPU_TS   ts_delta;
    CPU_TS   ts_delta_dly;
    CPU_TS   ts_delta_timeout;
```

```
    CPU_SR_ALLOC();
    (void)&p_arg;                                           //防止编译出错
    while (DEF_ON) {
        (void)OSTaskSemPend((OS_TICK)0,
                            (OS_OPT)OS_OPT_PEND_BLOCKING,
                            (CPU_TS  *)0,
                            (OS_ERR  *)&err);               //等待任务信号量
        if (err == OS_ERR_NONE) {                           //没有错误发生
            OS_CRITICAL_ENTER();                            //进入临界区
            OSTickCtr++;                                    //系统时钟节拍计数增加
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
            TRACE_OS_TICK_INCREMENT(OSTickCtr);
#endif
            OS_CRITICAL_EXIT();                             //退出临界区
            ts_delta_dly = OS_TickListUpdateDly();          //更新延迟链表
            ts_delta_timeout = OS_TickListUpdateTimeout();  //更新超时链表
            ts_delta = ts_delta_dly + ts_delta_timeout;
            if (OSTickTaskTimeMax < ts_delta) {
                OSTickTaskTimeMax = ts_delta;
}
}
    }
}
```

6.3.3 节拍链表任务插入函数

```
void  OS_TickListInsert(OS_TICK_LIST   *p_list,
                        OS_TCB         *p_tcb,
                        OS_TICK         time)
```

参数描述：

(1) p_list：指向要插入的链表头指针；

(2) p_tcb：指向要插入链表的 TCB 指针；

(3) time：任务就绪的剩余时间。

功能描述：

将任务插入到时钟节拍链表中。

```
void  OS_TickListInsert(OS_TICK_LIST   *p_list,
OS_TCB         *p_tcb,
OS_TICK         time)
{
    OS_TCB   *p_tcb1;
    OS_TCB   *p_tcb2;
    OS_TICK  remain;
```

```
    if (p_list->TCB_Ptr == (OS_TCB *)0) {                //时钟节拍链表为空
        p_tcb->TickRemain = time;                        //将 time 赋值给任务剩余时间
        p_tcb->TickNextPtr = (OS_TCB *)0;                //后继指针指向空
        p_tcb->TickPrevPtr = (OS_TCB *)0;                //前继指针指向空
        //任务时钟节拍链表指针指向 p_list
        p_tcb->TickListPtr = (OS_TICK_LIST *)p_list;
        p_list->TCB_Ptr = p_tcb;                 //时钟节拍链表指针的 TCB 指针指向 p_tcb
#if OS_CFG_DBG_EN > 0u
        p_list->NbrEntries = 1u;                         //时钟节拍链表个数等于 1
#endif
    } else {                                             //时钟节拍链表不为空
        p_tcb1 = p_list->TCB_Ptr;
        p_tcb2 = p_list->TCB_Ptr;
        remain = time;
        while (p_tcb2 != (OS_TCB *)0) {
            //任务剩余时间小于 p_tcb2 指向任务的剩余时间
            if (remain <= p_tcb2->TickRemain) {
                if (p_tcb2->TickPrevPtr == (OS_TCB *)0) {   //p_tcb2 前继指针为空
                    //将 remain 赋值给任务的剩余时间
     p_tcb->TickRemain = remain;
                    p_tcb 的前继指针指向空
            p_tcb->TickPrevPtr = (OS_TCB *)0;
                    //p_tcb 的后继指针指向 p_tcb2
            p_tcb->TickNextPtr = p_tcb2;
                    //p_tcb 的时钟节拍指针指向 p_list
           p_tcb->TickListPtr = (OS_TICK_LIST *)p_list;
                    //缩短 p_tcb2 的时间
                    p_tcb2->TickRemain -= remain;
                    //p_tcb2 的前继指针指向 p_tcb
                    p_tcb2->TickPrevPtr = p_tcb;
                    将 p_tcb 添加到任务链表中
                    p_list->TCB_Ptr = p_tcb;
#if OS_CFG_DBG_EN > 0u
                    p_list->NbrEntries++;                //链表中对象数目增加
#endif
                } else {
                    p_tcb1 = p_tcb2->TickPrevPtr;
             p_tcb->TickRemain = remain;                 //存储剩余时间
            p_tcb->TickPrevPtr = p_tcb1;
            p_tcb->TickNextPtr = p_tcb2;
                    //任务时钟节拍链表指针指向该链表
            p_tcb->TickListPtr = (OS_TICK_LIST *)p_list;
                    p_tcb2->TickRemain -= remain;
                    p_tcb2->TickPrevPtr = p_tcb;
                    p_tcb1->TickNextPtr = p_tcb;
#if OS_CFG_DBG_EN > 0u
```

```
                    p_list->NbrEntries++;
#endif
                }
                return;
            } else {
                remain -= p_tcb2->TickRemain;
                p_tcb1 = p_tcb2;
                p_tcb2 = p_tcb2->TickNextPtr;
            }
        }
        p_tcb->TickRemain = remain;              //p_tcb 剩余时间等于 remain
        p_tcb->TickPrevPtr = p_tcb1;             //p_tcb 的前继指针指向 p_tcb1
        p_tcb->TickNextPtr = (OS_TCB *)0;        //p_tcb 的后继指针指向空
        //p_tcb 的时钟节拍链表指针指向 p_list
        p_tcb->TickListPtr = (OS_TICK_LIST *)p_list;
        p_tcb1->TickNextPtr = p_tcb;             //p_tcb1 的后继指针指向 p_tcb
#if OS_CFG_DBG_EN > 0u
        p_list->NbrEntries++;                    //节拍链表中的对象数目加 1
#endif
    }
}
```

6.3.4 节拍链表任务删除函数

```
void  OS_TickListRemove(OS_TCB  *p_tcb)
```

参数描述：

p_tcb：指向要从链表中移出的 TCB 指针。

功能描述：

将任务从时钟节拍链表中移出。

```
void  OS_TickListRemove(OS_TCB  *p_tcb)
{
    OS_TICK_LIST  *p_list;
    OS_TCB        *p_tcb1;
    OS_TCB        *p_tcb2;

    p_list = (OS_TICK_LIST *)p_tcb->TickListPtr;
    p_tcb1 = p_tcb->TickPrevPtr;
    p_tcb2 = p_tcb->TickNextPtr;
    if (p_tcb1 == (OS_TCB *)0) {                         //p_tcb 的前继指针为空
        //p_tcb 的后继指针为空,即该时钟节拍链表中只有一个任务
        if (p_tcb2 == (OS_TCB *)0) {
```

```
                //清空该时钟节拍链表中的所有内容
                //时钟节拍链表的 TCB 指针指向空
                p_list->TCB_Ptr = (OS_TCB *)0;
#if OS_CFG_DBG_EN > 0u
                //时钟节拍链表的对象个数等于 0
                p_list->NbrEntries = (OS_OBJ_QTY)0u;
#endif
                p_tcb->TickRemain = (OS_TICK)0u;
                p_tcb->TickListPtr = (OS_TICK_LIST *)0;
            } else {//p_tcb 的后继指针不为空
                //p_tcb2 的前继指针指向空
                p_tcb2->TickPrevPtr = (OS_TCB *)0;
                p_tcb2->TickRemain += p_tcb->TickRemain;
                //时钟节拍链表的 TCB 指针指向 p_tcb2
                p_list->TCB_Ptr = p_tcb2;
#if OS_CFG_DBG_EN > 0u
                p_list->NbrEntries--;              //链表中对象个数减 1
#endif
                p_tcb->TickNextPtr = (OS_TCB *)0;
                p_tcb->TickRemain = (OS_TICK)0u;
                p_tcb->TickListPtr = (OS_TICK_LIST *)0;
            }
        } else {                                   //p_tcb 的前继指针不为空
 p_tcb1->TickNextPtr = p_tcb2;                     //p_tcb1 的后继指针指向 p_tcb2
            if (p_tcb2 != (OS_TCB *)0) {           //后继指针不为空
                p_tcb2->TickPrevPtr = p_tcb1;      //p_tcb2 的后继指针指向 p_tcb1
                //p_tcb2 的剩余时间等于本身剩余时间加上 p_tcb 的剩余时间
                p_tcb2->TickRemain += p_tcb->TickRemain;
            }
            p_tcb->TickPrevPtr = (OS_TCB *)0;      //p_tcb 的前继指针指向空
#if OS_CFG_DBG_EN > 0u
            p_list->NbrEntries--;
#endif
            p_tcb->TickNextPtr = (OS_TCB *)0;
            p_tcb->TickRemain = (OS_TICK)0u;
            p_tcb->TickListPtr = (OS_TICK_LIST *)0;
        }
}
```

6.4 时钟管理函数

6.4.1 延迟时钟节拍的延时函数

```
void  OSTimeDly(OS_TICK   dly,
OS_OPT    opt,
                 OS_ERR   *p_err )
```

延时模式：

该延时函数有三种延时模式，分别为 OS_OPT_TIME_DLY（相对模式），OS_OPT_TIME_MATCH（绝对模式），OS_OPT_TIME_PERIODIC（周期模式）。

参数说明：

（1）dly：延时的时钟节拍数，取值范围为 $0 \sim 2^{32}-1$。当 dly 为 0，任务没有延时，会报错 * p_err = OS_ERR_TIME_ZERO_DLY（在绝对模式下，dly 为 0 是合法的）。

（2）opt：指定延时模式，有 4 种取值：OS_OPT_TIME_DLY（默认模式）、OS_OPT_TIME_TIMEOUT、OS_OPT_TIME_MATCH、OS_OPT_TIME_PERIODIC，前两种为相对模式，第三种为绝对模式，最后一种为周期模式。

（3）p_err：指向函数返回的错误代码。

其中，p_err 可能包含以下几种情况：

① OS_ERR_NONE：函数被成功调用；

② OS_ERR_OPT_INVALID：该函数被指定了一个无效的 opt；

③ OS_ERR_SCHED_LOCKED：调度器被锁，不能执行延时函数；

④ OS_ERR_TIME_DLY_ISR：在中断服务程序中调用该函数；

⑤ OS_ERR_TIME_ZERO_DLY：指定延时（dly）为 0。

使用说明：

任务调用该函数实现延时 n 个时钟节拍，n 由 dly 和 opt 共同决定。调用该函数的任务会被挂起，插入到等待队列中。随后内核会进行调度，选择拥有最高优先级的任务继续运行。当延时时间结束或者其他任务调用了 OSTimeDlyResume() 函数后，原任务才会进入就绪状态。

相对模式是指相对当前时间，任务延时 n 个时钟节拍，当 OSTickCtr 等于之前的 OSTickCtr＋dly 时延时结束。在系统负荷较重时，延时可能会相差一个节拍，甚至相差两个节拍。

在周期模式下，任务虽然也有可能会被推迟执行，但是其平均效果要比相对模式好得多。因此，如果在系统中长时间使用延迟则应该使用周期模式。当 OSTickCtr 等于 OSTCBCurPtr－>TickCtrPrev＋dly 时，延时结束。

绝对模式通常是指机器在上电 n 个时钟节拍后执行某个特定的程序。当 OSTickCtr 等于 dly 时，延时结束。

该函数调用了 OS_CRITICAL_ENTER() 对临界区资源进行保护，调度器上锁；调用 OS_TickListInsert() 将任务加入时钟节拍列表 TickList 中；调用 OS_RdyListRemove() 将任务移出就绪队列；调用 OS_CRITICAL_EXIT_NO_SCHED() 退出临界区；调用 OSSched() 找到就绪列表中优先级最高的任务，执行上下文切换，如图 6.4 所示。

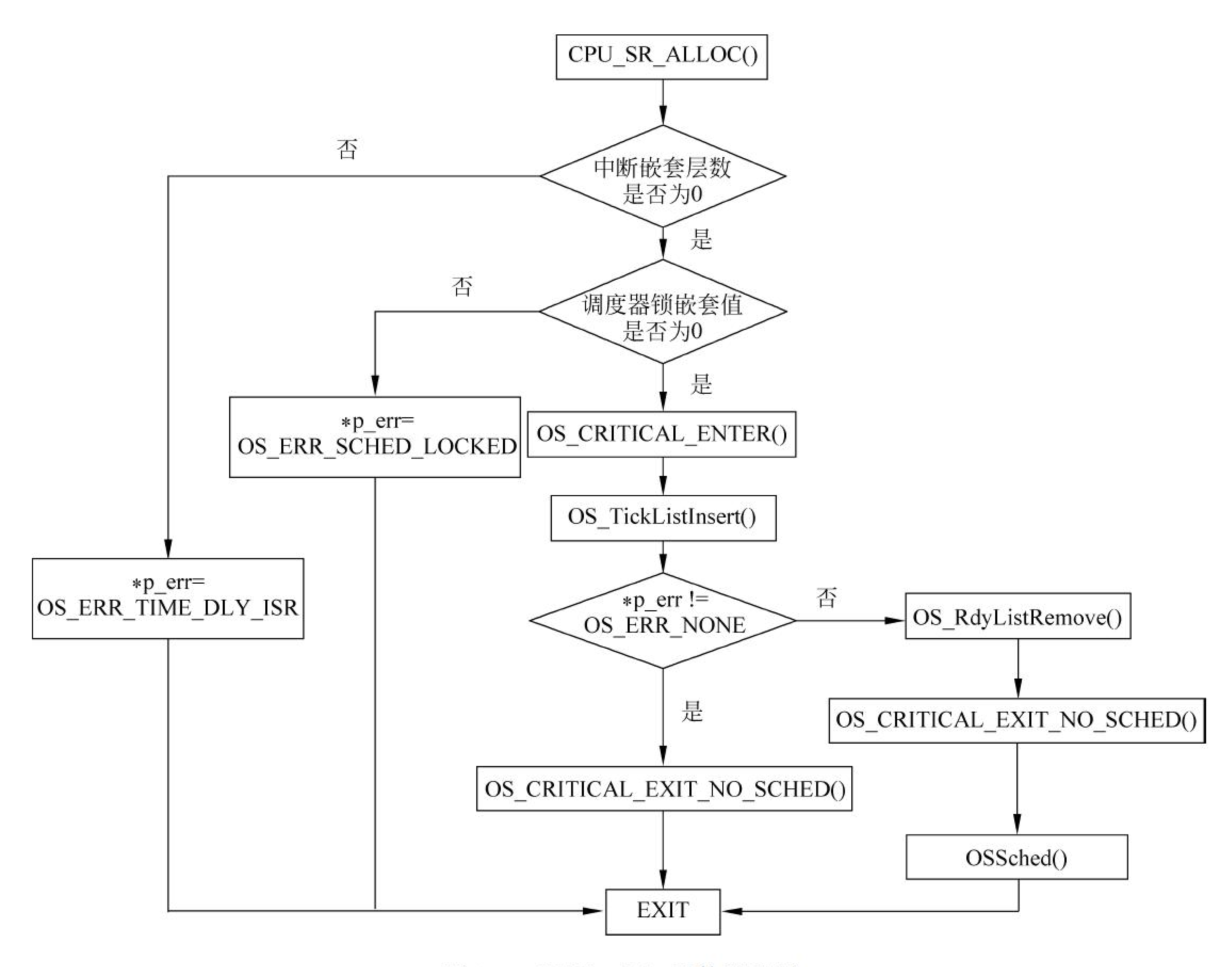


图 6.4 OSTimeDly 函数流程图

6.4.2 延迟具体时间的延时函数

```
void  OSTimeDlyHMSM(CPU_INT16U    hours,
    CPU_INT16U    minutes,
    CPU_INT16U    seconds,
                  CPU_INT32U    milli,
                  OS_OPT        opt,
                  OS_ERR       * p_err)
```

参数说明：

(1) hours：时，取值 0～99(对参数有严格要求)，0～99(对参数无严格要求)；

(2) minutes：分，取值 0～59(对参数有严格要求)，0～9999(对参数无严格要求)；

(3) seconds：秒，取值 0～59(对参数有严格要求)，0～65535(对参数无严格要求)；

(4) milli：毫秒，取值 0～999(对参数有严格要求)，0～4294967295(对参数无严格要求)；

(5) opt：OS_OPT_TIME_DLY(默认模式)、OS_OPT_TIME_TIMEOUT、OS_OPT_TIME_MATCH、OS_OPT_TIME_PERIODIC,前两种为相对模式,第三种为绝对模式,最后一种为周期模式,OS_OPT_TIME_HMSM_STRICT(默认模式,对参数有严格要求),OS_OPT_TIME_HMSM_NON_STRICT(对参数无严格要求);

(6) p_err：指向函数返回的错误代码。

其中,p_err 可能包含以下几种情况：

① OS_ERR_NONE：函数被成功调用;

② OS_ERR_OPT_INVALID：该函数被指定了一个无效的 opt;

③ OS_ERR_SCHED_LOCKED：调度器被锁,不能执行延时函数;

④ OS_ERR_TIME_DLY_ISR：在中断服务程序中调用该函数;

⑤ OS_ERR_TIME_INVALID_HOURS：指定一个无效的 hours 参数;

⑥ OS_ERR_TIME_INVALID_MINUTES：指定一个无效的 minutes 参数;

⑦ OS_ERR_TIME_INVALID_SECONDS：指定一个无效的 seconds 参数;

⑧ OS_ERR_TIME_INVALID_MILLISECONDS：指定一个无效的 milli 参数;

⑨ OS_ERR_TIME_ZERO_DLY：hours、minutes、seconds、milli 全部为 0。

使用说明：

该函数只在 OS_CFG_TIME_DLY_HMSM_EN=1 时才被系统创建,仅在相对模式下工作才有意义。用户指定的时间不能太长,不能超过 OS_TICK 类型变量所允许的最大值,应控制参数 hours、minutes 大小,milli 精度应为时钟频率的整数倍,否则会丢失精度。例如时钟频率为 100Hz,那么时钟节拍为 10ms,若延时设置为 5ms,则实际延时时间为 10ms,因为延时的精度小于时钟节拍的精度,当延时时间结束时,内核不能及时发现,会造成精度丢失。

该函数的主体代码与 OSTimeDly 大体一致,只是增加了对时间参数的正确性检验以及将时间(时、分、秒、毫秒)转换为时钟节拍数的过程。

6.4.3　延时取消函数

```
void  OSTimeDlyResume(OS_TCB   *p_tcb, OS_ERR   *p_err)
```

参数说明：

(1) p_tcb：指向要被唤醒任务的 OS_TCB 指针;

(2) p_err：指向函数返回的错误代码。

其中,p_err 可能包含以下几种情况：

① OS_ERR_NONE：任务被成功唤醒;

② OS_ERR_STATE_INVALID：任务处于无效的状态;

③ OS_ERR_TIME_DLY_RESUME_ISR：在中断服务程序中调用该函数;

④ OS_ERR_TIME_NOT_DLY：任务没有等待时间到期;

⑤ OS_ERR_TASK_SUSPENDED：任务不能够被唤醒，因为任务是被 OSTaskSuspend()函数挂起的。

使用说明：

该函数只有在 OS_CFG_TIME_DLY_RESUME_EN=1 时才能被系统创建，用来恢复其他调用了延时函数(OSTimeDly()、OSTimeDlyHMSM())的任务，该函数在 os_time.c 中定义只能被任务调用。该函数只能唤醒处于延时状态的任务(p_tcb－＞TaskState ＝＝ OS_TASK_STATE_DLY|OS_TASK_STATE_DLY_SUSPENDED)，如果要唤醒其他状态(等待、超时、挂起等)下的任务会报错(＊p_err ＝ OS_ERR_TASK_NOT_DLY)。被恢复的任务并不知道自己是被 OSTimeDlyResume()函数恢复的，它认为自己是延时期满而结束延时的。

该函数调用了 OS_TickListRemove()将任务从未就绪队列中移出，调用 OS_RdyListInsert()将任务加入就绪队列中，如图 6.5 所示。

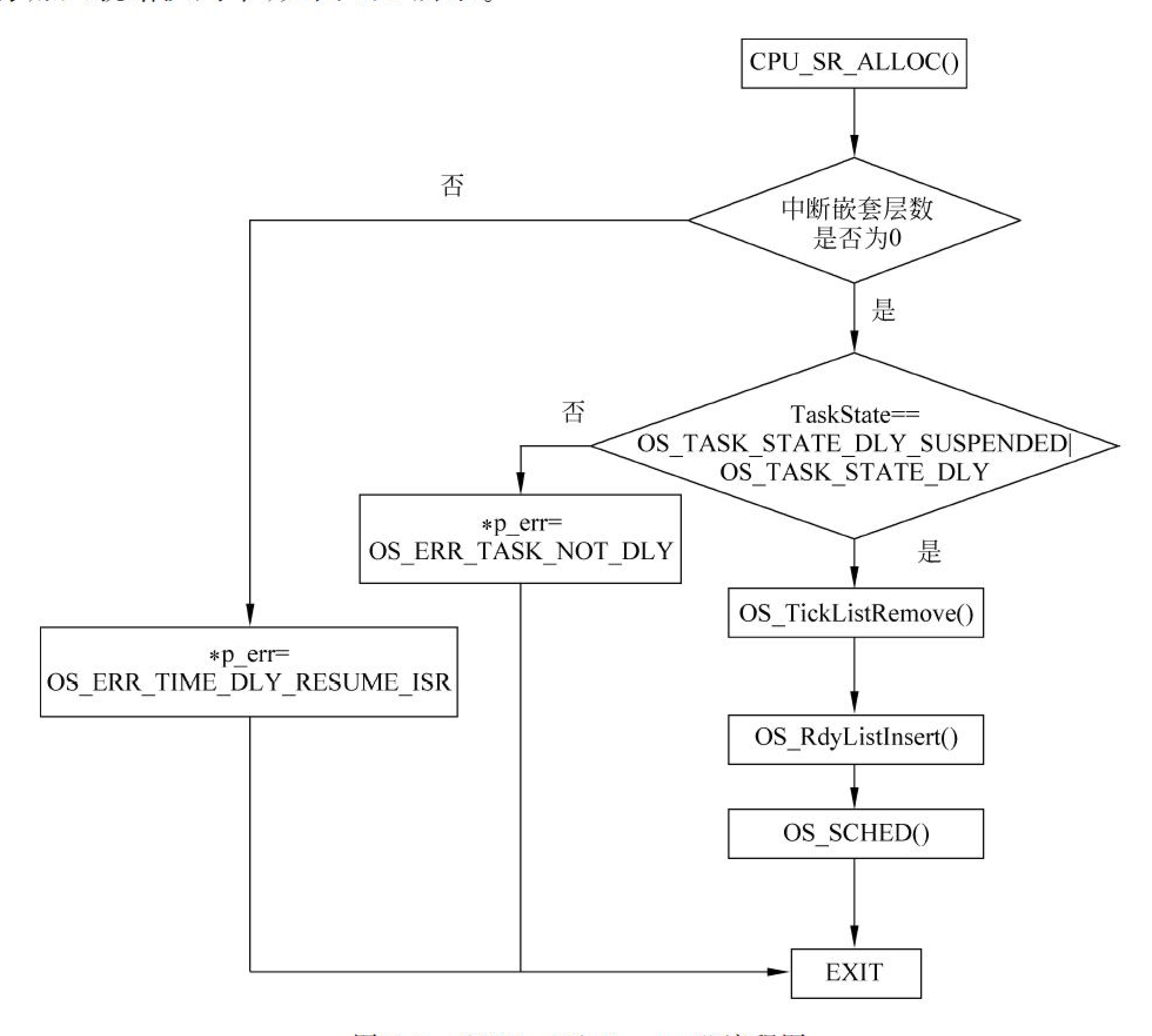


图 6.5 OSTimeDlyResume()流程图

注意：当任务状态 TaskState 为 OS_TASK_STATE_DLY_SUSPENDED 时，只会调用 OS_TickListRemove()将任务从时钟节拍列表中移出，并不会调用 OS_RdyListInsert()

将任务加入到就绪队列中，所以此时任务仍然处于 OS_TASK_STATE_SUSPENDED 挂起状态，不是就绪状态。

6.4.4 时钟节拍设置函数

```
OS_TICK  OSTimeGet(OS_ERR  *p_err)
```

参数说明：

p_err：指向函数返回的错误代码。

使用说明：

该函数被应用程序用来获取延时函数剩余的节拍数。

```
OS_TICK  OSTimeGet(OS_ERR  *p_err)
{
    OS_TICK  ticks;
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR *)0) {
        OS_SAFETY_CRITICAL_EXCEPTION();
        return ((OS_TICK)0);
    }
#endif

    CPU_CRITICAL_ENTER();
    ticks = OSTickCtr;                     //获取剩余的节拍数
    CPU_CRITICAL_EXIT();
   *p_err = OS_ERR_NONE;                   //设置错误标记代码为没有错误
    return (ticks);                        //返回剩余的节拍数
}
```

6.4.5 时钟节拍设置函数

```
void  OSTimeSet(OS_TICK   ticks,
                OS_ERR   *p_err)
```

参数说明：

（1）ticks：要设置的时钟节拍值；

（2）p_err：指向函数返回的错误代码。

使用说明：该函数被应用程序用来设置时钟节拍值。

```
void  OSTimeSet(OS_TICK   ticks,
                OS_ERR    * p_err)
{
    CPU_SR_ALLOC();

#ifdef OS_SAFETY_CRITICAL
    if (p_err == (OS_ERR * )0) {
         OS_SAFETY_CRITICAL_EXCEPTION();
         return;
    }
#endif

    CPU_CRITICAL_ENTER();
    OSTickCtr = ticks;              //设置时钟节拍
    CPU_CRITICAL_EXIT();
   * p_err      = OS_ERR_NONE;      //设置错误标记代码为没有错误
}
```

6.5 时钟管理应用

6.5.1 场景描述

现实生活中小区或者停车场的入口有许多栏杆，当汽车进入停车场时，栏杆每次只允许一辆车通过。当汽车通过时，安保人员会发给每辆车一个门卡，用于标识通过的车辆。现在模拟三辆汽车要进入停车场的情况，假设第一辆汽车在门口等待5s，其他汽车只有在前一辆车进入停车场后才能进入。图6.6展示了任务的主要逻辑关系。

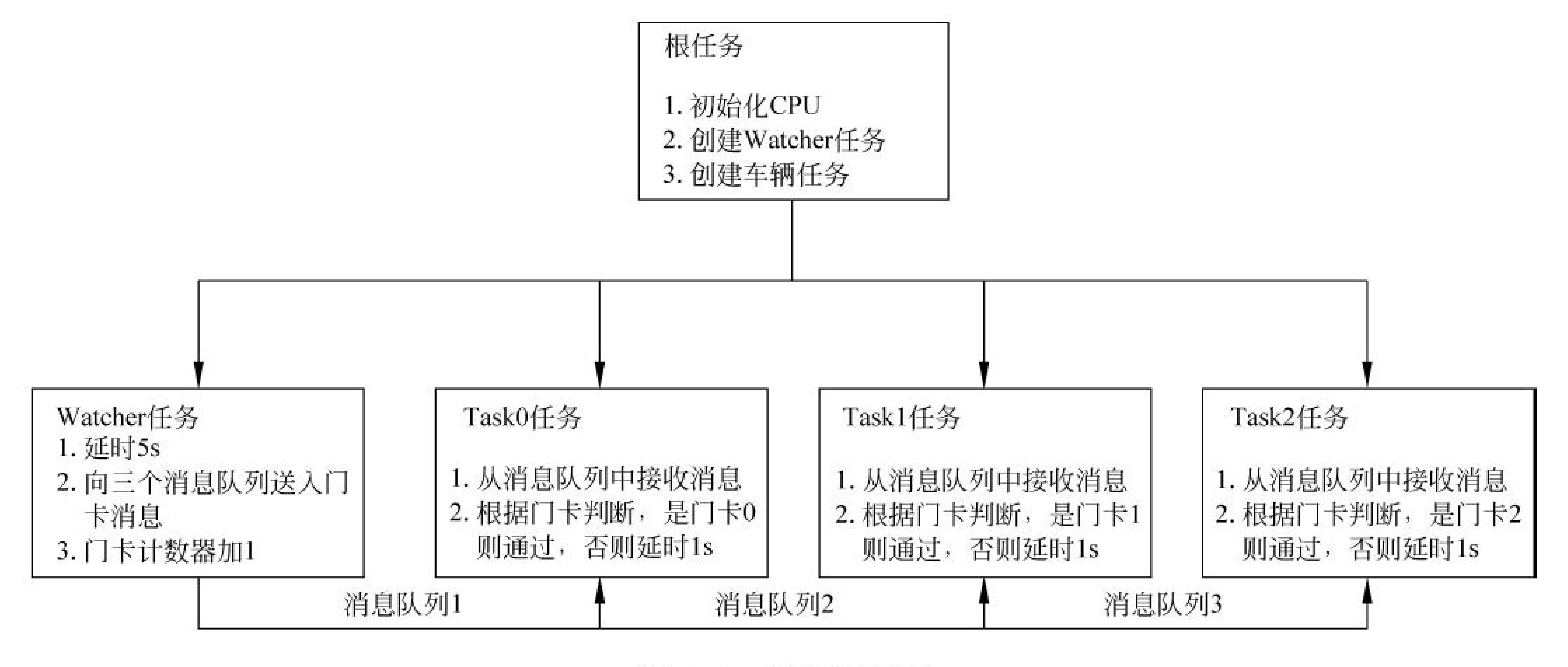


图6.6 任务逻辑图

6.5.2 运行环境

运行环境：软件平台 VS2013。

6.5.3 具体实现

app_cfg.h：

```
定义各个任务的优先级:
#define  APP_TASK_START_PRIO4u    //定义任务的优先级
#define  Watcher_PRIO6u
#define  TASK0_PRIO7u
#define  TASK1_PRIO                    7u
#define  TASK2_PRIO                    7u
定义任务栈的大小:
#define  APP_TASK_START_STK_SIZE1024u

#define  WatcherStkLength               128u
#define  Task0StkLength                 128u
#define  Task1StkLength                 128u
#define  Task2StkLength                 128u
```

app.c：

定义各个任务 TCB，创建任务栈，初始化用于令牌分发的 Counter 和三个消息队列指针。

```
static  OS_TCB MainTaskStartTCB;                //创建任务块
static  OS_TCB WatcherTCB;
static  OS_TCB Task0TCB;
static  OS_TCB Task1TCB;
static  OS_TCB Task2TCB;
//创建任务栈
static  CPU_STK MainTaskStartStk[APP_TASK_START_STK_SIZE];
static  CPU_STK WatcherStk[WatcherStkLength];
static  CPU_STK Task0Stk[Task0StkLength];
static  CPU_STK Task1Stk[Task1StkLength];
static  CPU_STK Task2Stk[Task2StkLength];

static int counter = 0;                         //初始计数器,用于门卡分发

static OS_Q Watcher_Q0;                         //消息队列
static OS_Q Watcher_Q1;
static OS_Q Watcher_Q2;
```

（1）声明任务函数如下所示。

```
static void MainTaskStart(void * p_arg);
static void Watcher(void * p_arg);
static void Task0(void * p_arg);
static void Task1(void * p_arg);
static void Task2(void * p_arg);
```

（2）在 main 函数中初始化系统，创建三个用于与车辆任务通信的消息队列，创建开始任务，之后启动多任务调用。

```
int main(void){
    OS_ERR err;
    OSInit(&err);

    OSQCreate(&Watcher_Q0, "Watcher_Q", 3, &err);
    OSQCreate(&Watcher_Q1, "Watcher_Q", 3, &err);
    OSQCreate(&Watcher_Q2, "Watcher_Q", 3, &err);

    OSTaskCreate((OS_TCB * )&MainTaskStartTCB,
                 (CPU_CHAR * )"Main Task Start",
                 (OS_TASK_PTR)MainTaskStart,
                 (void * ) 0,
                 (OS_PRIO)APP_TASK_START_PRIO,
                 (CPU_STK * )&MainTaskStartStk[0],
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE/10u,
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
                 (OS_MSG_QTY)0u,
                 (OS_TICK)0u,
                 (void * )0,
                 (OS_OPT)(OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR),
                 (OS_ERR * )&err);

                 OSStart(&err);
}
```

（3）在 MainTaskStart()中创建 Watcher 任务、Task0 任务、Task1 任务、Task2 任务。

```
static void MainTaskStart(void * p_arg){
      OS_ERR err;
      (void)p_arg;
      BSP_Init();
      CPU_Init();
#if OS_CFG_STAT_TASK_EN > 0u
     OSStatTaskCPUUsageInit(&err);
```

```
#endif
    APP_TRACE_DBG(("任务开始了...\n\r"));

    OSTaskCreate((OS_TCB *)&WatcherTCB,
        (CPU_CHAR *)"Watcher",
        (OS_TASK_PTR)Watcher,
        (void *)0,
        (OS_PRIO)Watcher_PRIO,
        (CPU_STK *)&WatcherStk[0],
        (CPU_STK_SIZE)WatcherStkLength / 10u,
        (CPU_STK_SIZE)WatcherStkLength,
        (OS_MSG_QTY)0u,
        (OS_TICK)0u,
        (void *)0,
        (OS_OPT)(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
(OS_ERR *)&err);
                    OSTaskCreate();          //创建 Task0
                    OSTaskCreate();          //创建 Task1
                    OSTaskCreate();          //创建 Task2
}
```

(4) Watcher 任务实现：使用 OSTimeDlyHMSM()延时函数，每隔 5s 向三个消息队列中发送消息，即门卡(0：允许 Task0 通过；1：允许 Task1 通过；2：允许 Task2 通过)。

```
static void Watcher(void *p_arg){
    OS_ERR err;

    APP_TRACE_DBG(("Watcher task created \n\r"));
    while (DEF_ON)
    {
        OSTimeDlyHMSM(0, 0, 5, 0, OS_OPT_TIME_DLY, &err);
        OSQPost(&Watcher_Q0, (void *)(counter % 3), sizeof(counter % 3),
                            OS_OPT_POST_FIFO, &err);
        OSQPost(&Watcher_Q1, (void *)(counter % 3), sizeof(counter % 3),
                            OS_OPT_POST_FIFO, &err);
        OSQPost(&Watcher_Q2, (void *)(counter % 3), sizeof(counter % 3),
                            OS_OPT_POST_FIFO, &err);
        APP_TRACE_DBG(("Watcher 发送令牌%d\n",counter%3));
        counter++;
    }
}
```

(5) Task0 任务实现：从消息队列中得到门卡，检查是否允许自己通过，如果是则输出“正在通过”，否则调用 OSTimeDlyHMSM()延时函数延时 1s。Task1 和 Task2 的实现与 Task0 相似。

```
static void Task0(void * p_arg){
    OS_ERR err;

    OS_MSG_SIZE nMsgSize = 0;
    int pMsg;
    CPU_TS nMsgTS;
    while (DEF_ON)
    {
        pMsg = (int)((char * )OSQPend(&Watcher_Q0, 0,
              OS_OPT_PEND_BLOCKING, &nMsgSize, &nMsgTS, &err));
        if (pMsg == 0)
            APP_TRACE_DBG(("Task0 正在通过...\n"));
        else{
            APP_TRACE_DBG(("Task0 停车等待...\n"));
            //停车等待 1s
            OSTimeDlyHMSM(0, 0, 1, 0, OS_OPT_TIME_DLY, &err);
        }
    }
}
```

6.5.4 实验结果

图 6.7 是程序的运行结果图，Watcher 任务在进程创建 5s 开始运行，依次给等待车辆发放门卡，等待车辆接收到门卡后进行匹配，如果是自己的门卡则通过，否则停车等待。

```
任务开始了...
Watcher task created
Watcher 发送令牌0
Task2 停车等待...
Task1 停车等待...
Task0 正在通过...
Watcher 发送令牌1
Task2 停车等待...
Task1 正在通过...
Task0 停车等待...
Watcher 发送令牌2
Task2 正在通过...
Task1 停车等待...
Task0 停车等待...
Watcher 发送令牌0
Task2 停车等待...
Task1 停车等待...
Task0 正在通过...
```

图 6.7 运行结果图

习题

1. 操作系统为什么需要时钟机制？
2. 请简要说明 μC/OS-Ⅲ的时钟机制。
3. 任务控制块 TCB 中哪些成员与时钟机制相关？
4. 延时任务列表是怎样初始化和形成的？
5. 时钟节拍任务是如何产生时钟中断的？
6. 节拍链表是如何插入和删除任务的？
7. 任务延时函数 OSTimeDly()和 OSTimeDlyHMSM()的区别是什么？
8. 延时取消函数 OSTimeDlyResume()是如何取消延时的？
9. 请详述时钟节拍获取函数和时钟节拍设置函数的内部机制。

第 7 章

定时器管理



7.1 定时器机制

定时器是操作系统中不可缺少的重要组成部分。应用程序可以通过定时器使进程在特定的时间执行特定的操作，使应用程序功能更加强大，更加灵活。μC/OS 操作系统在 μC/OS-Ⅱ 2.83 及其之后的版本中，引入了对软件定时器的支持。因此，μC/OS-Ⅲ也对定时器进行了支持。定时器的引入，使得 μC/OS 实时操作系统的功能更加完善。在实时操作系统中，一个优秀的软件定时器实现要求较高的精度，较小的处理器开销，并且占用较少的存储器资源。μC/OS-Ⅲ定时器的实现，是非常值得实时操作系统借鉴学习的。

在 μC/OS-Ⅲ操作系统内部，任务的延时功能和软件定时器功能，都需要底层硬件计数器的支持。μC/OS-Ⅲ操作系统软件定时器的实现，实质上就是实现了一个递增计数器。通过赋予定时器一个定时值和一个初始值为 0 的计数器，每隔一段时间，计数器值自动加 1，当计数器值的值等于定时器的值时，可以触发执行特定的操作。该操作由回调函数实现，应用程序可以根据自身功能需求实现相关的回调函数操作。回调函数是指通过函数指针调用的函数，是用户自己定义的，可以是简单地打开 LED 灯操作，或者开启电机等。回调函数执行时所用到的堆栈是定时器的任务堆栈，所以要确保分配的定时器任务堆栈大小能够满足回调函数的堆栈要求。回调函数的执行是根据它们在定时器链表中的位置先后执行，并且一个定时器只能执行一个回调函数。

定时器任务的执行时间极大程度上是由触发的时钟节拍个数和回调函数的执行时间决定。回调函数执行期间，调度处于被锁状态，所以回调函数执行越快越好，要避免在回调函数中等待。

在内存允许的情况下，应用程序可以定义多个定时器。当定时器定时完成时，回调函数会被立即调用执行，但一定不要在回调函数中使用阻塞调用或者可以阻塞或删除定时器任务的函数。

μC/OS-Ⅲ的时间系统是由 SysTick 提供的，因此定时器的时基也是由 SysTick 提供的，不同的是 μC/OS-Ⅲ的时钟节拍和定时器的时钟节拍是不一样的，μC/OS-Ⅲ的时钟节拍由 OS_CFG_TICK_RATE_HZ 决定，而定时器的时钟节拍由 OS_CFG_TMR_TASK_RATE_HZ 决定。通常来说，定时器时钟节拍都要比时钟节拍慢很多。如果定时器时钟节拍是 10Hz，而系统时钟节拍则可能是 1000Hz，那么节拍中断服务程序执行 100 次后，会使

定时器计数加1。定时器定时到期后，执行用户定义的回调函数。图7.1是定时器系统工作机制的流程图。

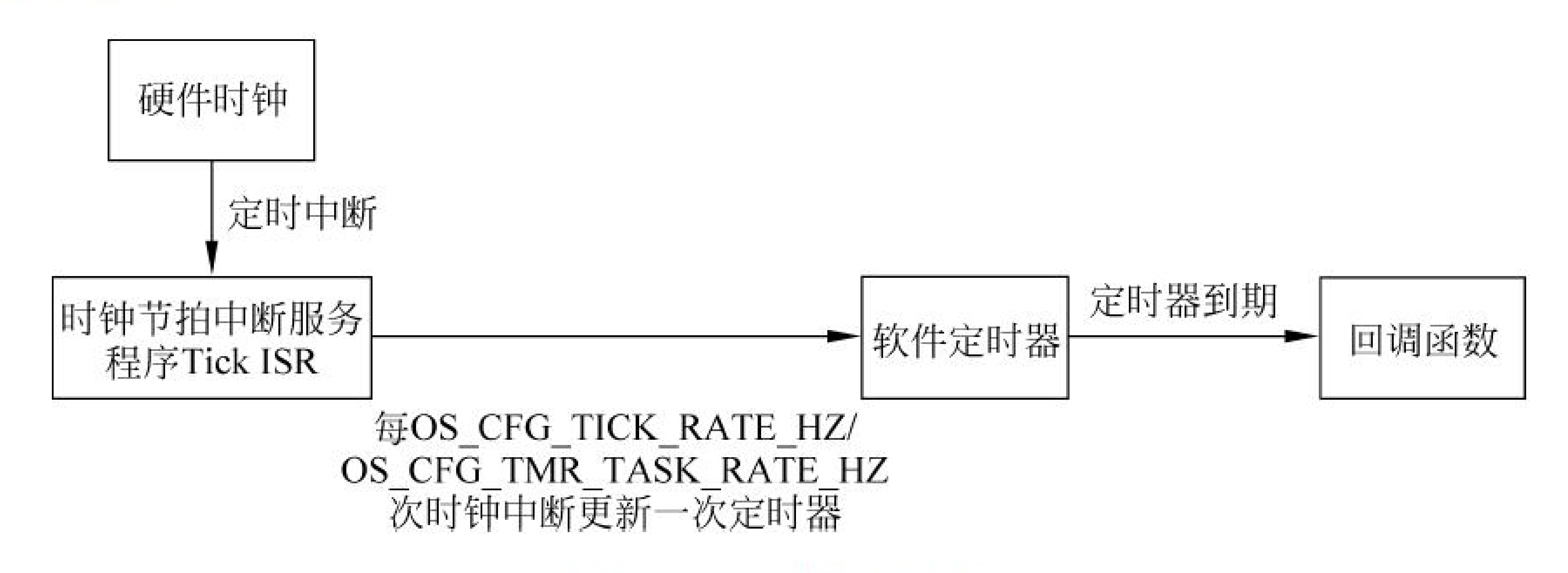


图7.1 定时器工作流程

7.2 定时器内部机制

7.2.1 定时器状态

μC/OS-Ⅲ在定时器的管理上，采用了定时器列表管理机制。定时器在μC/OS-Ⅲ中有四种状态：未使用状态(OS_TMR_STATE_UNUSED)、停止状态(OS_TMR_STATE_STOPPED)、运行状态(OS_TMR_STATE_RUNNING)和完成状态(OS_TMR_STATE_COMPLETED)。可以通过OSTmrStateGet()函数查看所创建的定时器所处的状态。“未使用状态”表示该定时器还未被创建或者已经被删除，无法被系统识别；“停止状态”表示调用OSTmrCreate()函数创建了定时器但还未使用OSTmrStart()函数来启动定时器或者调用了OSTmrStop()函数；“运行状态”表示调用了OSTmrStart()函数后的定时器状态，在此状态时，定时器被删除、定时器停止或者单次运行结束，都会改变该状态；“完成状态”是单次定时器独有的状态。图7.2是定时器的状态转换图。

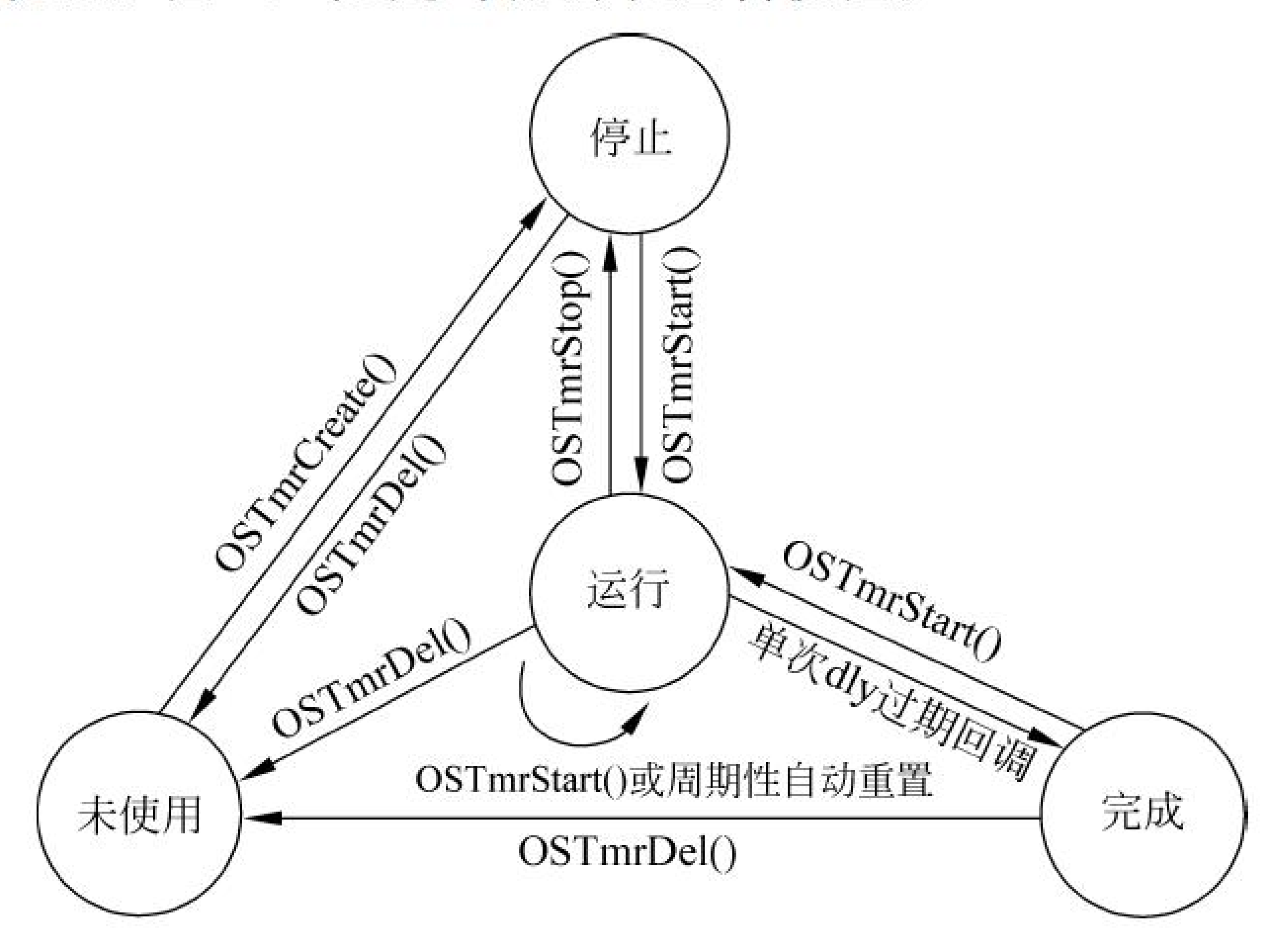


图7.2 定时器状态转换图

7.2.2 定时器结构体 os_tmr

```
struct  os_tmr {
    #if OS_OBJ_TYPE_REQ > 0u
        OS_OBJ_TYPE            Type;
    #endif
    #if OS_CFG_DBG_EN > 0u
        CPU_CHAR               *NamePtr;            //定时器的名字
    #endif
        OS_TMR_CALLBACK_PTR    CallbackPtr;         //定时器到时的回调函数
        void                   *CallbackPtrArg;     //定时器过期时传递函数的参数
        OS_TMR                 *NextPtr;            //双向链表指针
        OS_TMR                 *PrevPtr;
        OS_TICK                Remain;              //定时器超时前的剩余时间
        OS_TICK                Dly;                 //延时
        OS_TICK                Period;              //周期
        OS_OPT                 Opt;                 //功能选择
        OS_STATE               State;
    #if OS_CFG_DBG_EN > 0u
        OS_TMR                 *DbgPrevPtr;
        OS_TMR                 *DbgNextPtr;
    #endif
};
typedef struct os_tmr OS_TMR;
```

7.2.3 定时器分类

在创建定时器时，需要指定定时器的类别，在 OSTmrCreate()函数中指定 opt 参数。定时器分为 3 种，分别是单次定时器、无初始延迟周期定时器和有初始延迟周期定时器。

单次定时器，顾名思义，定时器从开始到结束只能被调用一次。定时器创建成功后，使用 OSTmrStart()函数启动定时器，期间可以使用 OSTmrStart()函数重启定时器，重置定时器计数时间，也可以通过 OSTmrStop()函数停止定时器。

无初始延迟周期定时器，定时器被设置为周期模式。定时器初始化过程中，设置延迟为0(即 dly=0)，定时器全程使用 period 参数指定的时钟节拍作为计数周期。定时器定时完成后，调用回调函数，同时使用 period 参数指定的值重置自己，开启下一个定时，一直循环下去。在定时器执行过程中，可以使用 OSTmrStart()函数重启定时器。

有初始延迟周期定时器，定时器被设置为周期模式。定时器初始化过程中，设置延时参数 dly 初始值，定时器初次延时是 dly 参数指定的节拍数。之后定时器延时节拍数由 period 参数指定。在定时器执行过程中，可以使用 OSTmrStart()函数重启定时器。

7.2.4 定时器管理时序

软件定时器同样由 SysTick 提供时钟，但是软件定时器的时钟是由 OS_CFG_TMR_TASK_RATE_HZ 决定的。也就是在 μC/OS-Ⅲ的时钟节拍上再做一次“分频”，软件定时

器的最快时钟节拍等于 μC/OS-Ⅲ的系统时钟节拍。这也就决定了软件定时器的精确度。节拍中断服务程序 ISR 负责每节拍(时钟频率/定时器任务频率)发送一个信号给定时器任务,其时序图如图 7.3 所示。

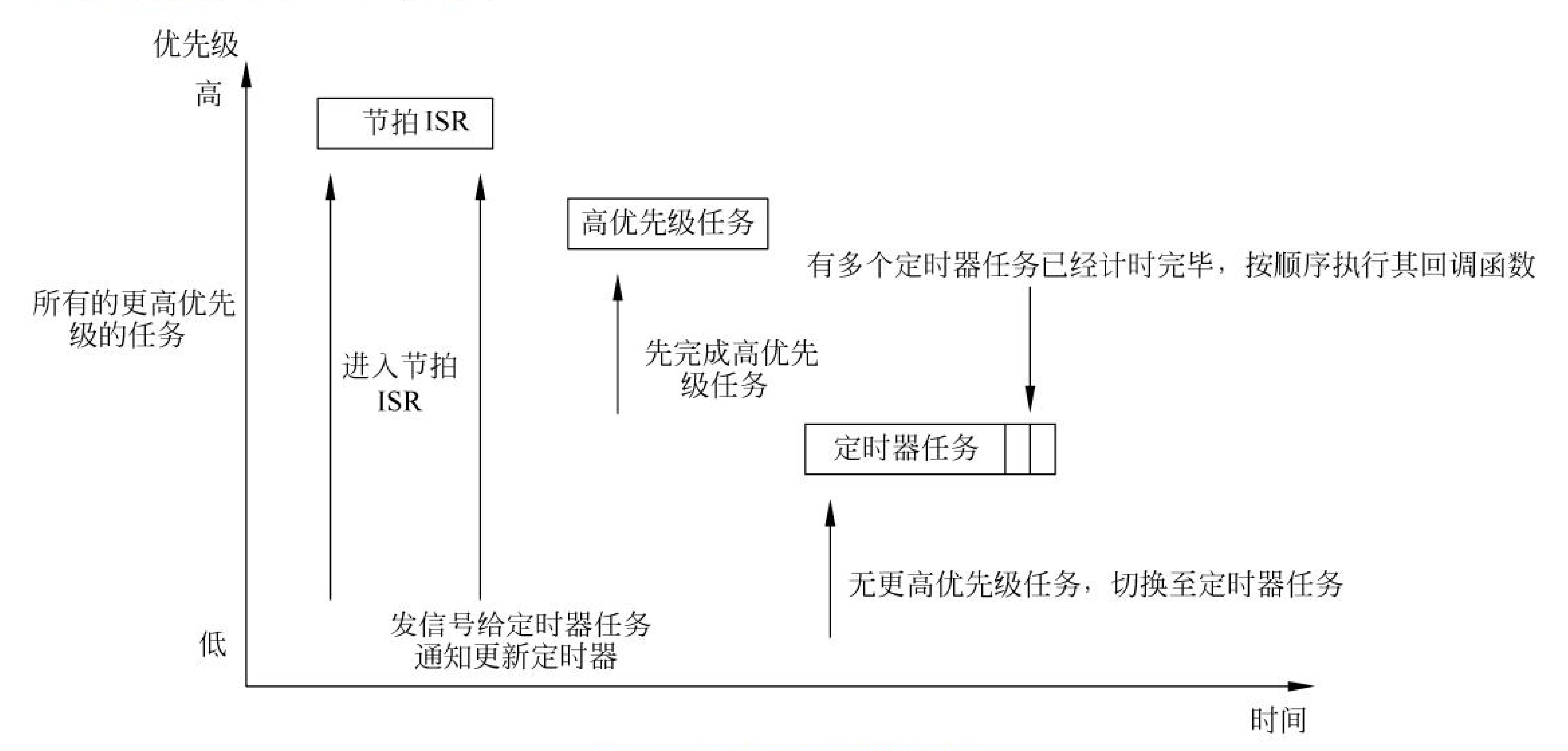


图 7.3　定时器管理时序图

7.2.5　软件定时器的实现原理

μC/OS-Ⅲ中软件定时器的实现方法是将定时器按定时时间分组,使得每次时钟节拍到来时只对部分定时器进行比较操作,缩短了每次处理的时间。但这就需要动态维护一个定时器组。定时器组的维护只是在每次定时器到时才发生,而且定时器从组中移除和再插入操作不需要排序。这是一种高效的算法,减少了维护所需的操作时间,其实现流程如图 7.4 所示。

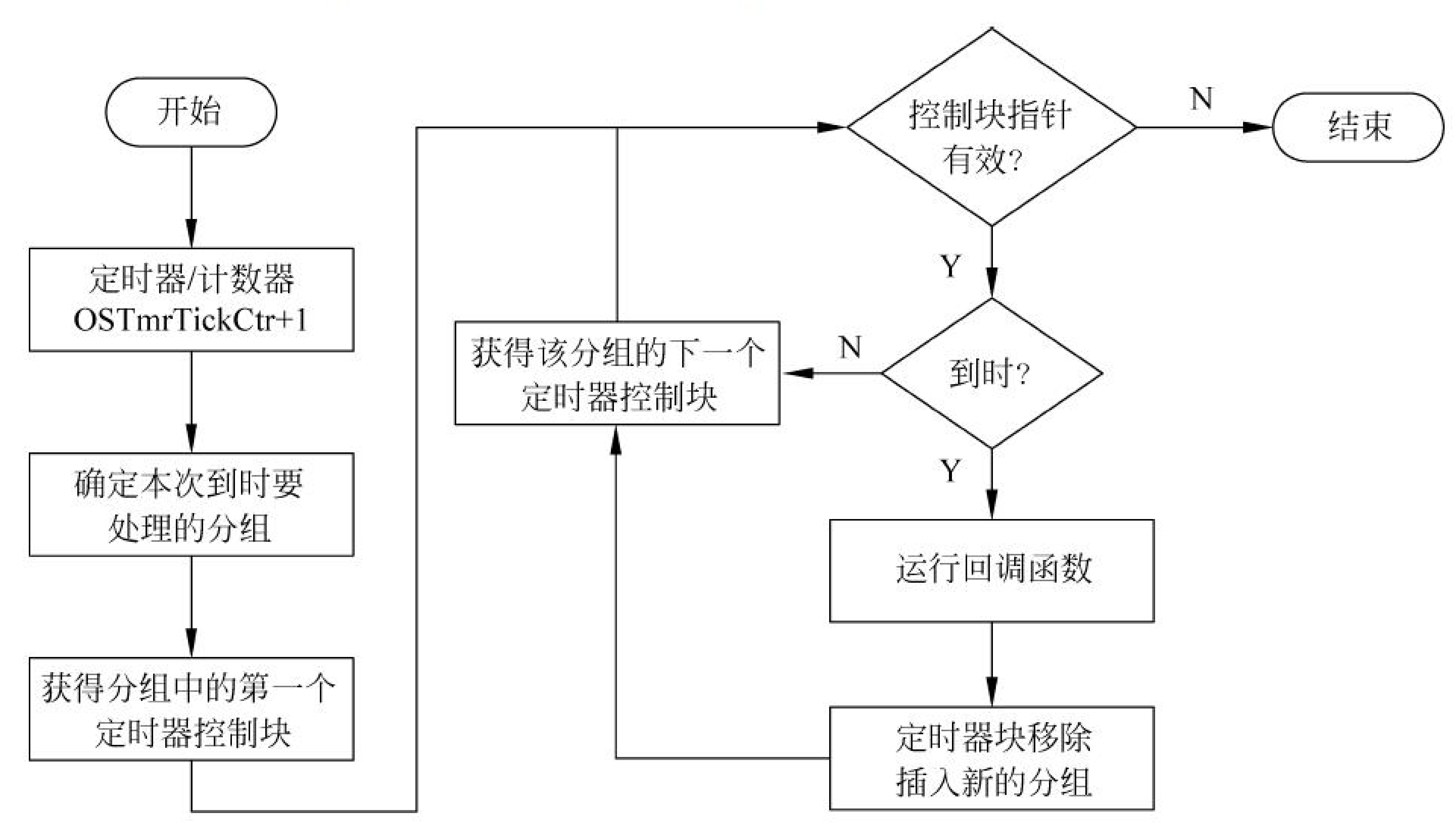


图 7.4　定时器任务管理流程图

7.2.6 主要的数据结构分析

无论是应用程序还是内核程序，其实都是对某些数据结构的增删改查之类的操作，所以一个系统设计的好坏，始于数据结构的定义，要弄清楚一个系统，也应该从最底层的数据结构着手。图 7.5 便是定时器数据结构如何从硬件一步一步实现上层软件定时器的功能示意图。

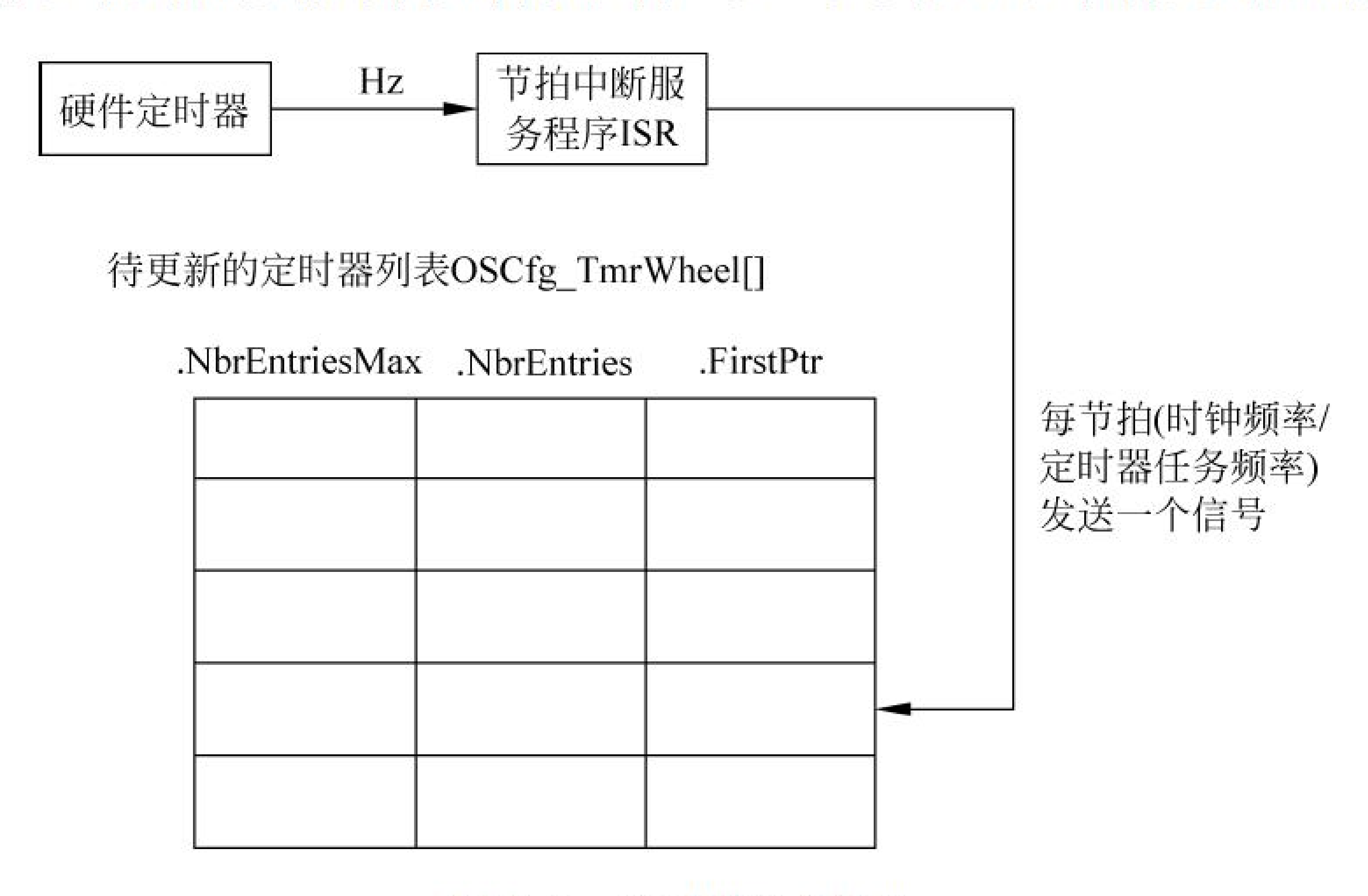


图 7.5 定时器任务管理

μC/OS-Ⅲ通过一个计数器 OSTmrTickCtr 和一个数组 OSCfg_TmrWheel[]来管理定时器。OSCfg_TmrWheel[]数组的每一项组成如图 7.5 中所示，分别是 NbrEntriesMax、NbrEntries 和 FirstPtr。NbrEntriesMax 表示定时器链表中定时器的最大数量，NbrEntries 表示定时器链表中已有定时器的数量，FirstPtr 表示指向定时器链表中第一个定时器的指针。

当新创建的定时器插入到 OSCfg_TmrWheel[]数组中时，需要判断插入到哪一个定时器链表中，μC/OS-Ⅲ计算待插入链表公式如下：

（1）MatchValue＝OSTmrTickCtr＋dly；

（2）OSCfg_TmrWheel[]中的序号＝MatchValue % OS_CFG_TMR_WHEEL_SIZE。

其中，MatchValue 即为 OS_TMR 中的 Match，dly 为定时器延时节拍数，OS_CFG_TMR_WHEEL_SIZE 为 OSCfg_TmrWheel[]数组的大小。

当 ISR 发送一个信号，OSTmrTickCtr 加 1，对应数组中第 OSTmrTickCtr % OSCfg_TmrWheel[OS_CFG_TMR_WHEEL_SIZE]条目若存在，则将第一个定时器任务的 Match 值与 OSTmrTickCtr 值相比较，若匹配，则将其移除，并且检查下一个，然后调用回调函数。如果不匹配，则结束查找，因为同一条目下的链表已经排序，按照剩余时间少的在前，剩余时间多的在后。定时器列表可以抽象成一个定时器轮，如图 7.6 所示。

μC/OS-Ⅲ的定时器列表管理与时钟节拍列表类似。这里为了方便理解，将 OSCfg_TmrWheel[]抽象为一个转轮。

OSCfg_TmrWheel[OS_TMR_CFG_WHEEL_SIZE]数组的每个元素都是已开启定时

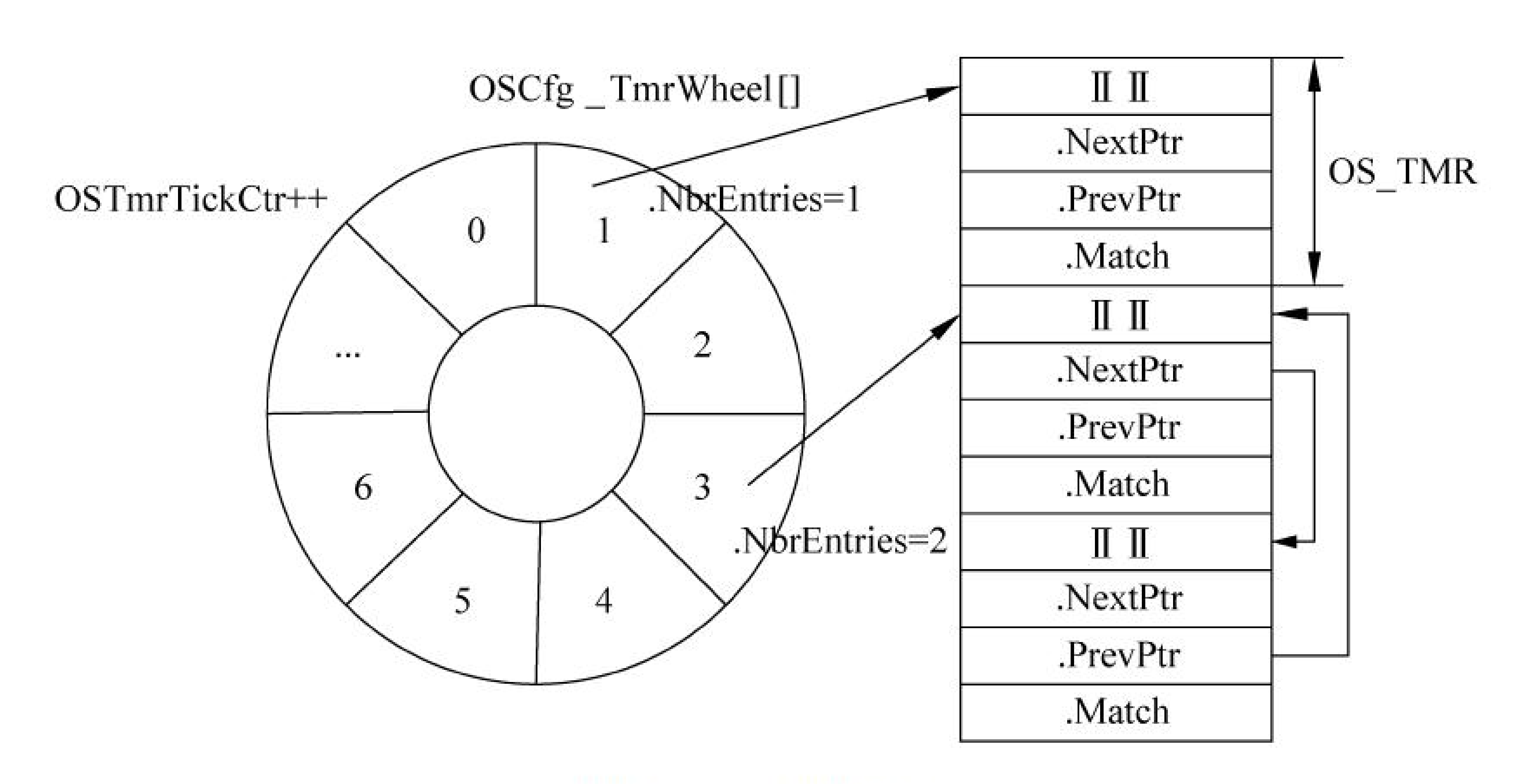


图 7.6 定时器列表

器的一个分组,元素中记录了指向该分组中第一个定时器控制块的指针,以及.NbrEntries 定时器控制块的个数。

7.3 定时器函数

7.3.1 定时器创建函数

```
void  OSTmrCreate(OS_TMR                  *p_tmr,
                  CPU_CHAR                *p_name,
                  OS_TICK                 dly,
                  OS_TICK                 period,
                  OS_OPT                  opt,
                  OS_TMR_CALLBACK_PTR     p_callback,
                  void                    *p_callback_arg,
                  OS_ERR                  *p_err)
```

参数说明:

(1) p_tmr: 定时器控制块的指针;

(2) p_name: 指定定时器的名称,常在调试时使用该变量;

(3) dly: 一次性延时时表示延时时间,周期性延时时表示进入周期延时前的第一次延时时间;

(4) period: 周期性延时时表示周期性延时的时间;

(5) opt: 表示此定时器的类型。其中,OS_TMR_OPT_ONE_SHOT 表示是一次性延时,OS_TMR_OPT_PERIOD 表示周期性延迟;

(6) p_callback: 回调函数;

(7) p_callback_arg: 回调函数的参数;

(8) p_err：返回状态，失败则返回错误状态，成功则返回定时器的句柄。

其中，p_err 可能包含以下几种情况：

① OS_ERR_NONE：定时器创建成功；

② OS_ERR_ILLEGAL_CREATE_RUN_TIME：非法创建运行时定时器错误，在调用了函数 OSSafetyCriticalStart()之后尝试创建定时器；

③ OS_ERR_OBJ_CREATED：定时器已经被创建；

④ OS_ERR_OBJ_PTR_NULL：p_tmr 是一个空指针；

⑤ OS_ERR_OBJ_TYPE：对象类型无效；

⑥ OS_ERR_OPT_INVALID：指定了一个无效的 opt；

⑦ OS_ERR_TMR_INVALID_DLY：指定了一个无效的延迟 delay；

⑧ OS_ERR_TMR_INVALID_PERIOD：指定了一个无效的 period；

⑨ OS_ERR_TMR_ISR：在中断程序中调用该函数。

使用说明：

OSTmrCreate()函数创建了一个定时器，指定了定时器的定时模式和定时器的定时时间。当定时器指定的时间到达时，调用用户定义的回调函数进行相关的处理。图 7.7 显示了 OSTmrCreate()函数的执行流程。

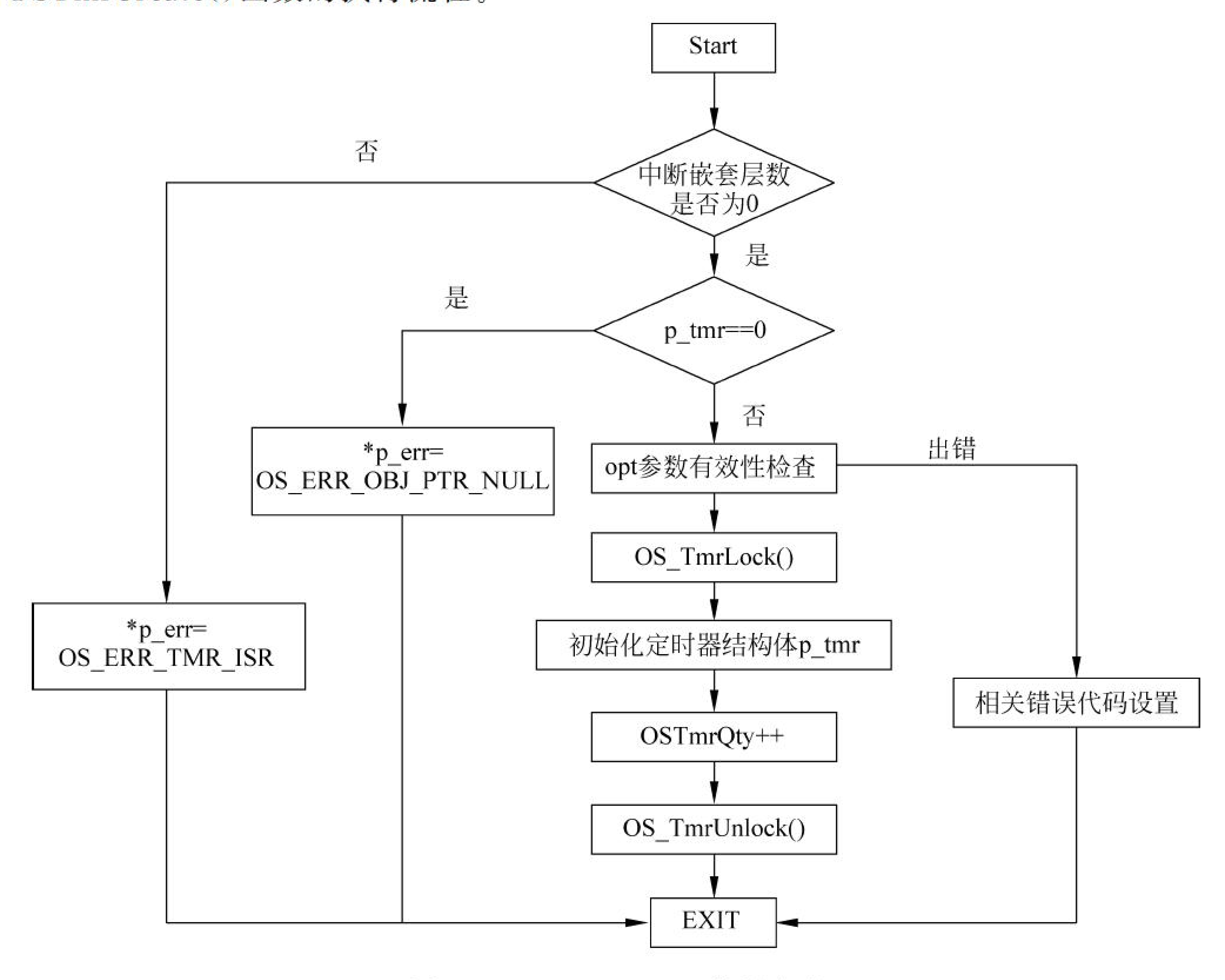


图 7.7 OSTmrCreate()执行流程

需要留意的是，OSTmrCreate()仅仅是创建了一个定时器，并没有启动定时器。可以通过调用 OSTmrStart()启动定时器。

7.3.2 定时器删除函数

```
CPU_BOOLEAN  OSTmrDel(OS_TMR  *p_tmr,  OS_ERR  *p_err)
```

参数说明：

(1) p_tmr：定时器控制块的指针；

(2) p_err：错误码，其值有以下几种可能。

① OS_ERR_NONE；

② OS_ERR_OBJ_TYPE：第一个参数未指向一个定时器类型；

③ OS_ERR_TMR_INVALID：第一个参数 p_tmr 是空指针；

④ OS_ERR_TMR_ISR：函数被 ISR 调用；

⑤ OS_ERR_TMR_INACTIVE：第一个参数指向的定时器未被创建；

⑥ OS_ERR_TMR_INVALID_STATE：第一个参数指向的定时器是无效状态。

返回值：(1) DEF_TRUE：定时器被成功删除；

(2) DEF_FALSE：定时器删除过程中出现了错误，没有被删除。

使用说明：

定时器删除成功则返回真，删除失败则返回假。OSTmrDel()具体流程图如图 7.8 所示。

其中删除处理具体代码如下：

```
switch (p_tmr->State) {
    case OS_TMR_STATE_RUNNING:            //定时器处于运行状态
         OS_TmrUnlink(p_tmr);             //把定时器从运行队列中移除
         OS_TmrClr(p_tmr);                //清空 p_tmr 指向的结构体
         OS_TmrUnlock();                  //解锁
         OSTmrQty--;                      //定时器数目减 1
         *p_err = OS_ERR_NONE;
         success = DEF_TRUE;
         break;

    case OS_TMR_STATE_STOPPED:            //定时器处于停止状态
    case OS_TMR_STATE_COMPLETED:          //定时器处于完成状态
         OS_TmrClr(p_tmr);
         OS_TmrUnlock();
         OSTmrQty--;
         *p_err = OS_ERR_NONE;
         success = DEF_TRUE;
         break;
```

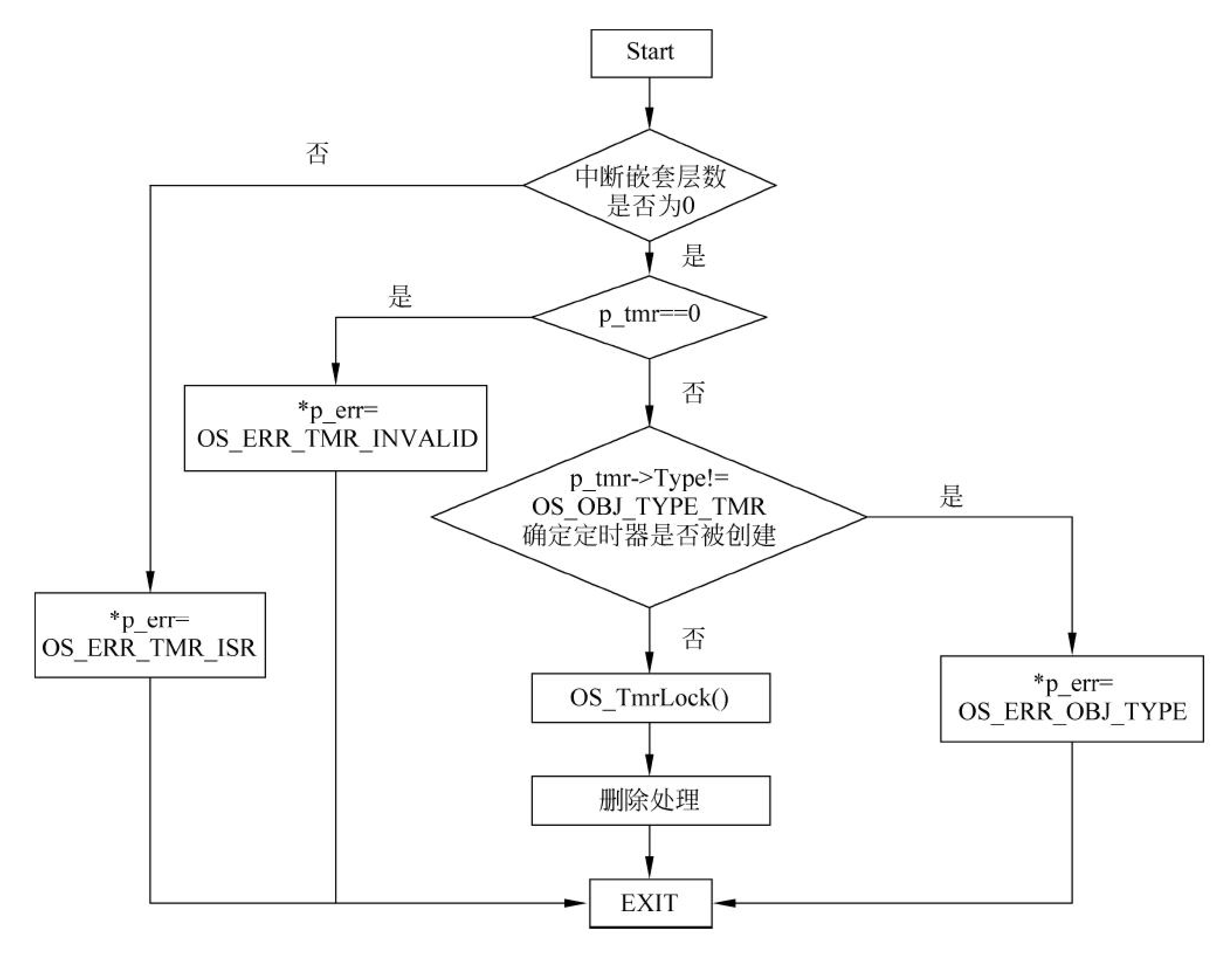


图 7.8 OSTmrDel()流程图

```
    case OS_TMR_STATE_UNUSED:     //定时器处于未使用状态,已被删除
         OS_TmrUnlock();
         *p_err = OS_ERR_TMR_INACTIVE;
         success = DEF_FALSE;
         break;

    default:
         OS_TmrUnlock();
         *p_err = OS_ERR_TMR_INVALID_STATE;
         success = DEF_FALSE;
         break;
}
```

7.3.3 获取定时器的剩余时间

```
OS_TICK  OSTmrRemainGet(OS_TMR  *p_tmr,  OS_ERR  *p_err)
```

参数说明：

(1) p_tmr：指向定时器控制块的指针；

(2) p_err：错误码，其值有以下几种可能。

① OS_ERR_NONE；

② OS_ERR_OBJ_TYPE：第一个参数未指向一个定时器类型；

③ OS_ERR_TMR_INVALID：第一个参数是空指针；

④ OS_ERR_TMR_ISR：函数被 ISR 调用；

⑤ OS_ERR_TMR_INACTIVE：第一个参数指向的定时器未被创建；

⑥ OS_ERR_TMR_INVALID_STATE：第一个参数指向的定时器是无效状态。

使用说明：

(1) 该函数返回一个定时器超时前所剩余的时间。

(2) 该函数的参数检验流程和 OSTmrDel() 函数极其相似，可以参考图 7.8 的描述。这里只给出获取定时器剩余时间的关键代码。

```
switch (p_tmr->State) {
    case OS_TMR_STATE_RUNNING:                  //定时器处于运行状态
         remain = p_tmr->Remain;                //获取定时器的剩余时间
         *p_err = OS_ERR_NONE;
         break;

    case OS_TMR_STATE_STOPPED:                  //定时器处于停止状态
          //定时器是周期性定时器
         if (p_tmr->Opt == OS_OPT_TMR_PERIODIC) {
if (p_tmr->Dly == 0u) {                         //无初始延迟
                remain = p_tmr->Period;         //剩余时间等于周期延时
} else {                                        //有初始延迟
                remain = p_tmr->Dly;            //剩余时间等于初始延时
}
} else {                                        //定时器是单次定时器
remain = p_tmr->Dly;                            //剩余时间为设置的延时节拍
}
         *p_err = OS_ERR_NONE;
         break;

case OS_TMR_STATE_COMPLETED:                    //定时器为完成状态
         *p_err = OS_ERR_NONE;
         remain = (OS_TICK)0;                   //剩余时间为 0
         break;

    case OS_TMR_STATE_UNUSED:                   //定时器为未使用状态
         *p_err = OS_ERR_TMR_INACTIVE;
         remain = (OS_TICK)0;                   //剩余时间为 0
```

```
        break;
    default:
        * p_err = OS_ERR_TMR_INVALID_STATE;
        remain = (OS_TICK)0;          //剩余时间为 0
        break;
}
```

7.3.4 定时器启动

```
CPU_BOOLEAN  OSTmrStart (OS_TMR  * p_tmr,  OS_ERR  * p_err)
```

参数说明：

(1) p_tmr：指向定时器的指针；

(2) p_err：错误码，与函数 OSTmrRemainGet()中的第二个参数有相同含义。

使用说明：

(1) 调用该函数用来启动对应的定时器。

(2) 该函数的参数检验流程和 OSTmrDel()函数极其相似，可以参考图 7.8 的描述。这里只给出定时器启动的关键代码。

```
switch (p_tmr->State) {
    case OS_TMR_STATE_RUNNING:                    //定时器处于运行状态
    CPU_CRITICAL_ENTER();                         //进入临界区
    p_tmr->Remain = p_tmr->Dly;                   //定时器剩余时间为定时器延时
    CPU_CRITICAL_EXIT();                          //退出临界区
    * p_err = OS_ERR_NONE;
    success = DEF_TRUE;
    break;

    case OS_TMR_STATE_STOPPED:                    //定时器处于停止状态
    case OS_TMR_STATE_COMPLETED:                  //定时器处于完成状态
    OS_TmrLock();                                 //定时器上锁
        //设置定时器状态为运行状态
    p_tmr->State = OS_TMR_STATE_RUNNING;
        //定时器初始延迟节拍为 0
        if (p_tmr->Dly == (OS_TICK)0) {
        //定时器剩余时间设置为周期节拍
    p_tmr->Remain = p_tmr->Period;
        } else {                                  //定时器初始延迟节拍不为 0
            //定时器剩余时间设置为节拍延迟
    p_tmr->Remain = p_tmr->Dly;
    }
    if (OSTmrListPtr == (OS_TMR *)0) {            //定时器链表指针为空
```

```
        p_tmr->NextPtr = (OS_TMR *)0;
        p_tmr->PrevPtr = (OS_TMR *)0;
                //将该定时器设置为定时器链表头
        OSTmrListPtr = p_tmr;
        OSTmrListEntries = 1u;
        } else {                          //定时器链表指针不为空
        p_next = OSTmrListPtr;            //插入定时器链表
        p_tmr->NextPtr = OSTmrListPtr;
        p_tmr->PrevPtr = (OS_TMR *)0;
        p_next->PrevPtr = p_tmr;
                //将该定时器设置为定时器链表头
        OSTmrListPtr = p_tmr;
        OSTmrListEntries++;               //定时器数目加 1
        }
        OS_TmrUnlock();                   //定时器解锁
        *p_err = OS_ERR_NONE;
        success = DEF_TRUE;
        break;

        case OS_TMR_STATE_UNUSED:         //定时器处于未使用状态
            //定时器不能直接从未使用状态跳转到运行状态
        *p_err = OS_ERR_TMR_INACTIVE;
        success = DEF_FALSE;
        break;

        default:
        *p_err = OS_ERR_TMR_INVALID_STATE;
        success = DEF_FALSE;
        break;
    }
```

7.3.5 定时器状态获取函数

```
OS_STATE  OSTmrStateGet(OS_TMR  *p_tmr,  OS_ERR  *p_err)
```

参数说明：

(1) p_tmr：指向定时器控制块的指针；

(2) p_err：与定时器启动函数 OSTmrStart()中的第二个参数有相同的含义。

使用说明：

该函数用来确定定时器的当前状态。定时器的状态如下：

(1) OS_TMR_STATE_UNUSED：定时器还未创建；

(2) OS_TMR_STATE_STOPPED：定时器被创建，但还未被执行或者已经被停止；

(3) OS_TMR_STATE_COMPLETED：定时器是一个单次定时器，并且已经超时

完成；

(4) OS_TMR_SATE_RUNNING：定时器正在运行。

该函数的参数检验流程和 OSTmrDel()函数极其相似，可以参考图 7.8 的描述。这里只给出获取定时器状态的关键代码。

```
state = p_tmr -> State;          //获取定时器状态
switch (state) {
    case OS_TMR_STATE_UNUSED:
    case OS_TMR_STATE_STOPPED:
    case OS_TMR_STATE_COMPLETED:
    case OS_TMR_STATE_RUNNING:
     * p_err = OS_ERR_NONE;
    break;

default:
      * p_err = OS_ERR_TMR_INVALID_STATE;
     break;
}
return (state);                   //返回定时器状态
```

7.3.6 定时器停止函数

```
CPU_BOOLEAN  OSTmrStop (OS_TMR   * p_tmr,
      OS_OPT    opt,
      void     * p_callback_arg,
      OS_ERR   * p_err)
```

参数说明：

(1) p_tmr：指向定时器控制块的指针。

(2) opt：指定定时器被停止后要完成的动作，有如下三种选择：

① OS_OPT_TMR_NONE：定时器停止后不做任何动作；

② OS_OPT_TMR_CALLBACK：定时器停止后，执行定时器创建时指定的回调函数，并将定时器创建时指定的参数传给回调函数；

③ OS_OPT_TMR_CALLBACK_ARG：定时器停止后，执行定时器创建时指定的回调函数，并传递当前函数指定的参数。

(3) p_callback_arg：指向定时器回调函数的新参数。即在上述第二个参数 opt 是第三种情况 OS_OPT_TMR_CALLBACK_ARG 时，就会用该参数指定的参数内容代替定时器被创建时指定的参数。

(4) p_err：错误码，该参数与函数 OSTmrStateGet()的第二参数类似，不同之处是多了几种错误类型，具体如下：

① OS_ERR_OPT_INVALID：opt 参数类型指定错误；

② OS_ERR_TMR_NO_CALLBACK：定时器未指定回调函数；

③ OS_ERR_TMR_STOPPED：定时器已经是停止状态。

使用说明：

该函数的作用是将指定的定时器强行停止。若成功停止返回真，否则返回假。该函数的参数检验流程和 OSTmrDel()函数极其相似，可以参考图 7.8 的描述。这里只给出定时器停止功能的关键代码。

```
switch (p_tmr->State) {
    case OS_TMR_STATE_RUNNING:                    //定时器为运行状态
        OS_TmrLock();                             //定时器上锁
        OS_TmrUnlink(p_tmr);                      //将定时器从定时器链表中移除
        *p_err = OS_ERR_NONE;
        switch (opt) {
        //定时器停止后,依然执行回调函数,参数为定时器创建时指定的参数
        case OS_OPT_TMR_CALLBACK:
            p_fnct = p_tmr->CallbackPtr;
            if (p_fnct != (OS_TMR_CALLBACK_PTR)0) {
                (*p_fnct)((void *)p_tmr, p_tmr->CallbackPtrArg);
            } else {
                *p_err = OS_ERR_TMR_NO_CALLBACK;
            }
            break;
        //定时器停止后,依然执行回调函数,参数为当前函数指定的参数
        case OS_OPT_TMR_CALLBACK_ARG:
            p_fnct = p_tmr->CallbackPtr;
            if (p_fnct != (OS_TMR_CALLBACK_PTR)0) {
                (*p_fnct)((void *)p_tmr, p_callback_arg);
            } else {
                *p_err = OS_ERR_TMR_NO_CALLBACK;
            }
            break;
        case OS_OPT_TMR_NONE:                     //定时器停止后不做任何动作
             break;
        default:
            OS_TmrUnlock();
            *p_err = OS_ERR_OPT_INVALID;
            return (DEF_FALSE);
        }
        OS_TmrUnlock();
        success = DEF_TRUE;
    break;
    case OS_TMR_STATE_COMPLETED:                  //定时器处于完成状态
    case OS_TMR_STATE_STOPPED:                    //定时器处于停止状态
```

```
        * p_err = OS_ERR_TMR_STOPPED;
         success = DEF_TRUE;
         break;

    case OS_TMR_STATE_UNUSED:        //定时器处于未使用状态
        * p_err = OS_ERR_TMR_INACTIVE;
        success = DEF_FALSE;
        break;

    default:
        * p_err = OS_ERR_TMR_INVALID_STATE;
        success = DEF_FALSE;
        break;
}
```

7.4 应用实例

7.4.1 场景描述

该部分要根据特定的场景，然后利用定时器实现一个功能。这里要实现的是人在24小时内的作息时间提醒，当到达预设的时间时，发出提醒，告知当前时刻需要做的事情。为了短时间内验证实验的效果，将24小时替换为24秒。

7.4.2 设计过程

运行环境：软件平台VS2013。

根据场景描述，此实例需要创建1个主任务和11个分任务，每个任务都有相应的定时器来完成具体操作。

7.4.3 具体实现

app_cfg.h：

1. 定义各个任务的优先级

```
#define  APP_TASK_START_PRIO        4u    //定义任务的优先级
#define  Time_24_PRIO               4u
```

2. 定义任务栈的大小

```
#define  APP_TASK_START_STK_SIZE      1024u
#define  Time_24_STK_SIZE             1024u
```

app.c:

1. 定义各个任务 TCB、创建任务栈

```
static  OS_TCB MainTaskStartTCB;                //创建任务块
static  OS_TCB Time_24TCB;
//创建任务栈
static  CPU_STK MainTaskStartStk[APP_TASK_START_STK_SIZE];
static  CPU_STK Time_24Stk[Time_24_STK_SIZE];
```

2. 声明任务函数和回调函数

```
static void Time_24(void * p_arg);
//回调函数
void Get_Up(OS_TMR * p_tmr, void * p_arg);
void Have_Breakfast(OS_TMR * p_tmr, void * p_arg);
void Go_To_Class(OS_TMR * p_tmr, void * p_arg);
void Go_To_Another_Class(OS_TMR * p_tmr, void * p_arg);
void Rest(OS_TMR * p_tmr, void * p_arg);
void Have_Lunch(OS_TMR * p_tmr, void * p_arg);
void Take_Nap(OS_TMR * p_tmr, void * p_arg);
void Study(OS_TMR * p_tmr, void * p_arg);
void Have_Dinner(OS_TMR * p_tmr, void * p_arg);
void Play_Game(OS_TMR * p_tmr, void * p_arg);
void Sleep(OS_TMR * p_tmr, void * p_arg);
```

3. 在 main 函数中初始化系统，创建 Time_24 任务

```
int main(void){
    OS_ERR err;
    OSInit(&err);    //初始化 μC/OS-Ⅲ

    OSTaskCreate((OS_TCB *)&Time_24TCB,
                 (CPU_CHAR *)"aaa",
                 (OS_TASK_PTR)Time_24,
                 (void *) 0,
                 (OS_PRIO)Time_24_PRIO,
                 (CPU_STK *)&Time_24Stk[0],
                 (CPU_STK_SIZE)Time_24_STK_SIZE / 10u,
                 (CPU_STK_SIZE)Time_24_STK_SIZE,
                 (OS_MSG_QTY)0u,
                 (OS_TICK)0u,
                 (void *)0,
                 (OS_OPT)(OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR),
                 (OS_ERR *)&err);

    OSStart(&err);
}
```

4. 在 Time_24 任务中创建一天生活中的各个定时器

```
static void Time_24(void * p_arg){
    OS_ERR err;
    (void)p_arg;
    BSP_Init();     //初始化 BSP
    CPU_Init();     //初始化 CPU
    #if OS_CFG_STAT_TASK_EN > 0u
    OSStatTaskCPUUsageInit(&err);
    #endif

    printf("这是一个人正常作息时间定时器提醒系统\n");
    printf(" **************************************** \n");
    OSTimeDly(500, 0, &err);

    OSTmrCreate(&task1, "timer1", 70, 24000, OS_OPT_TMR_PERIODIC, Get_Up, 0, &err);

    if (OSTmrStart(&task1, &err))
    {
        printf("Get_Up 定时器创建成功   \n");
        printf(" **************************************** \n");
        OSTimeDly(500, 0, &err);
    }
    else
    {
        printf("创建定时器失败   \n");
        printf(" **************************************** ");
        OSTimeDly(500, 0, &err);
    }

    ……   //其他各个定时器的实现都非常类似
}
```

5. 实验结果

运行结果如图 7.9 所示。

```
这是一个人正常作息时间定时器提醒系统
****************************************
Get_Up定时器创建成功
****************************************
7点了，该起床了
****************************************
7点半了，该吃早饭了
****************************************
8点了，该上课了
****************************************
9点半了，该上另一节课了
****************************************
11点了，该休息了
****************************************
12点了，该吃午饭了
****************************************
12点半了，该小憩一会了
****************************************
13点了，该学习了
****************************************
18点了，该吃晚饭了
****************************************
17点了，该玩游戏了
****************************************
```

图 7.9　运行结果

习题

1. 为什么操作系统需要定时器机制?
2. 请简要说明 μC/OS-Ⅲ的定时器机制。
3. μC/OS-Ⅲ内核定时器有几种状态?每种状态代表什么含义?
4. 定时器状态转换是如何发生的?
5. 请详述定时器结构体 os_tmr 的内容。
6. μC/OS-Ⅲ定时器有哪些类型?
7. μC/OS-Ⅲ内核是如何进行定时器任务管理的?
8. 内核在创建定时器的过程中做了哪些主要的工作?
9. 内核在删除定时器的过程中做了哪些主要的工作?
10. 定时器是如何启动和停止的?

第8章 内存管理



8.1 内存管理机制

内存管理机制可以让操作系统更合理地利用有限的内存资源，保证程序的高效运行。优秀的操作系统必须能够对内存进行有效的管理，响应应用程序的请求，给应用程序提供分配和释放内存的服务。μC/OS-Ⅲ作为实时操作系统同样也实现了对内存的管理。

在应用程序开发过程中，某些开发语言（例如C语言，C++）是支持动态申请内存的，应用程序通过malloc/free系列内存管理函数对内存进行管理。当调用malloc()分配内存空间结束后，一定要调用free()释放申请的内存空间，防止出现内存泄露问题。同时，因为每次分配的内存都大小不一，多次分配会导致一块连续的内存被分成若干个内存小块，从而出现了内存碎片，进而使得操作系统的可用内存越来越小。此外，malloc/free系列函数执行时间不确定，因此实时操作系统通常都根据需要创建自己的内存管理机制。

μC/OS-Ⅲ中的内存管理主要采用内存分区控制块实现。μC/OS-Ⅲ将连续的内存区域划分为多个内存分区，内存分区内包含整数个大小相同的内存块，根据应用程序的请求分配和释放内存块。因为同一个内存分区内的内存块都已事先分配完毕，并且大小相等，所以不会产生内存碎片，并且申请和释放内存的时间也是固定的。内存分区布局如图8.1所示。

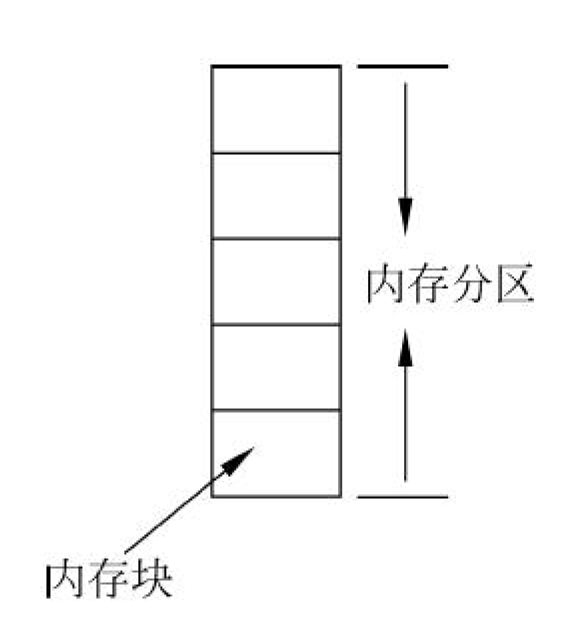


图8.1 内存分区布局

μC/OS-Ⅲ的内存分区有很大的灵活性。应用程序可以根据自身需求申请合适大小的内存分区，并且根据需要将内存分区划分为合适大小的内存块。当应用程序申请内存块时，可以直接从内存分区中获取（OSMemGet），内存块使用结束后，再把内存块释放（OSMemPut），把内存块归还到内存分区中。

8.2 内存管理机制分析

8.2.1 内存控制块 os_mem

μC/OS-Ⅲ使用内存控制块 os_mem 来管理内存分区,每一个结构体 os_mem 对应着一块内存分区。os_mem 结构体中记录了内存分区的重要内容,os_mem 的源码如下:

```
struct  os_mem {
    #if OS_OBJ_TYPE_REQ > 0u
    //内存控制块的类型,应该被设置为 OS_OBJ_TYPE_MEM
    OS_OBJ_TYPE     Type;
    #endif
    //内存分区起始地址指针
    void      *AddrPtr;
    #if OS_CFG_DBG_EN > 0u
    //内存分区名称
    CPU_CHAR      *NamePtr;
    #endif
    //空闲内存控制块列表指针
    void      *FreeListPtr;
    //每一个内存控制块的大小
    OS_MEM_SIZE      BlkSize;
    //分区中内存控制块的总数
    OS_MEM_QTY       NbrMax;
    //分区中空闲内存控制块总数
    OS_MEM_QTY       NbrFree;
    #if OS_CFG_DBG_EN > 0u
    //调试链表头指针
    OS_MEM      *DbgPrevPtr;
    //调试链表尾指针
    OS_MEM      *DbgNextPtr;
    #endif
};
```

内存分区结构如图 8.2 所示。

如果要在 μC/OS-Ⅲ中使用内存管理,需要在 os_cfg.h 文件中将开关量 OS_CFG_MEM_EN 设置为 1。这样 μC/OS-Ⅲ在启动时就会对内存管理器进行初始化(由 OSInit()调用 OSMemInit()实现)。

8.2.2 内存分区调试链表指针 OSMemDbgListPtr

μC/OS-Ⅲ内存管理系统定义了一个元素类型为 OS_MEM 的链表 OSMemDbgListPtr 来存放所有的 OS_MEM 结构体。当用户创建一个新内存分区时,则向该链表中加入一个 OS_MEM 结构体来管理新建立的分区。调试链表指针如图 8.3 所示。

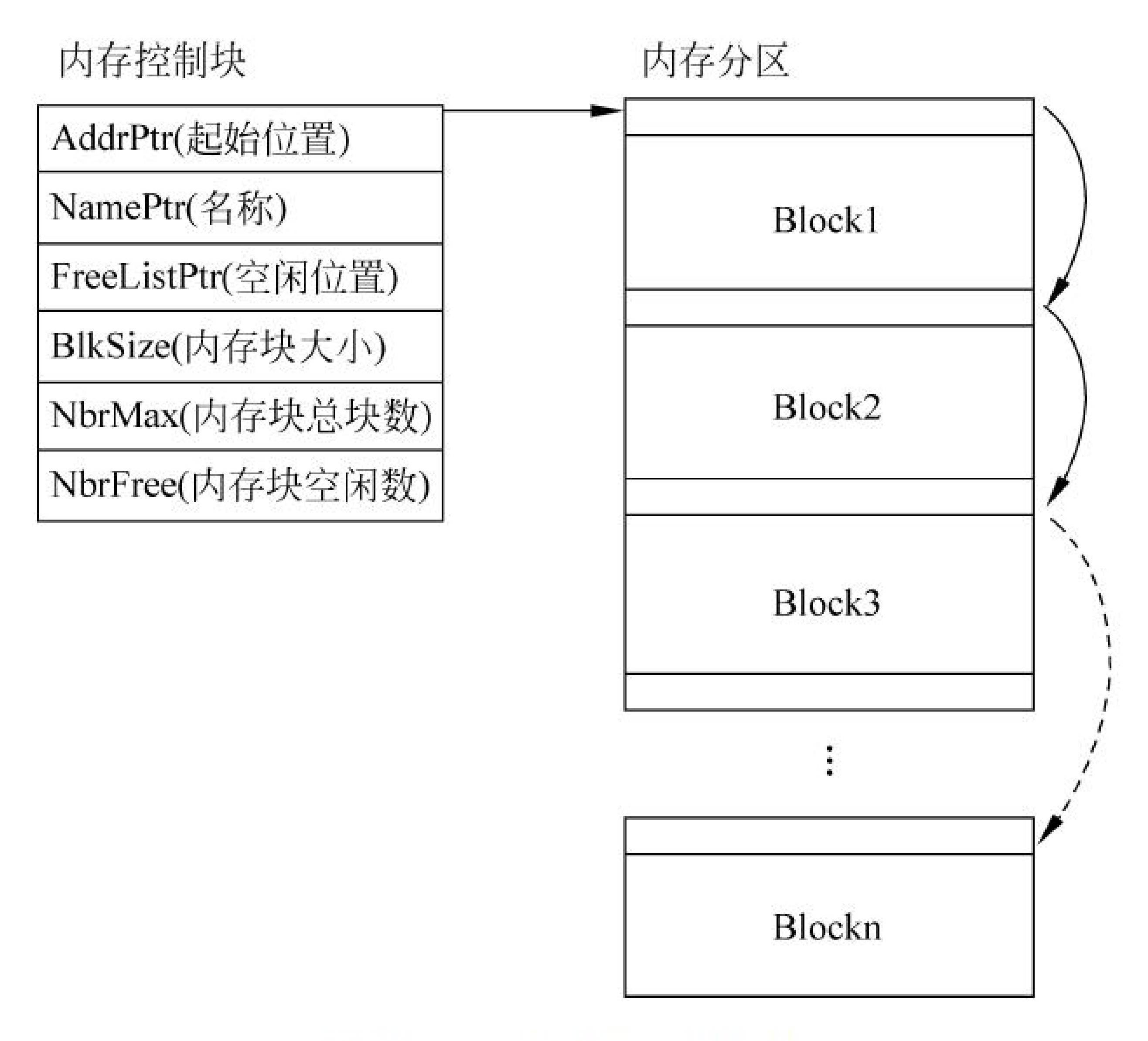


图 8.2 内存分区结构图

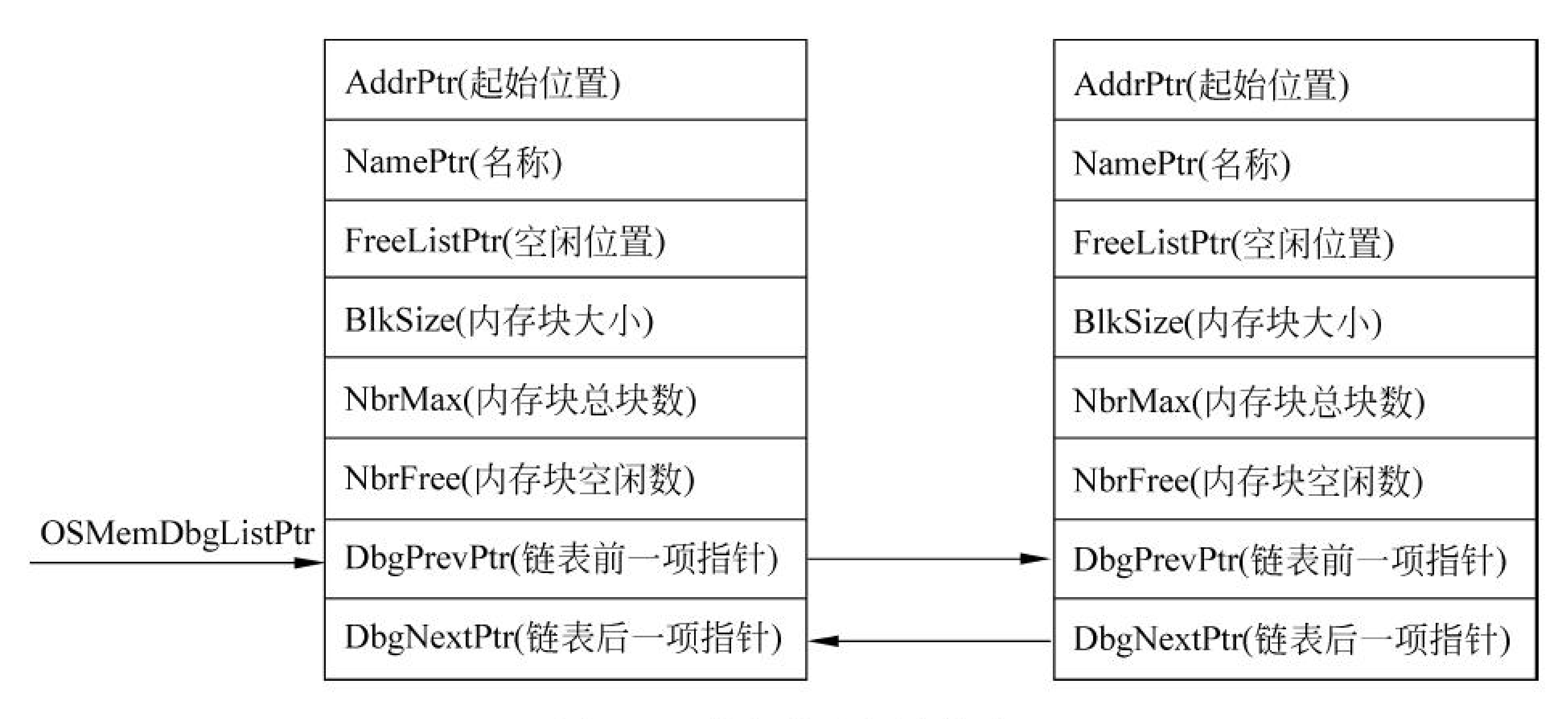


图 8.3 内存分区调试链表

8.3 内存管理函数

8.3.1 内存初始化函数

```
void  OS_MemInit(OS_ERR  *p_err)
```

参数说明：

p_err：指向函数返回的错误代码。

使用说明：

该函数被 μC/OS-Ⅲ调用，用于初始化内存分区管理器。该函数是 μC/OS-Ⅲ的内部函数，应用程序不能调用该函数。

该函数主要有两个功能，一是初始化内存调试列表，二是设置创建的内存分区管理器的数目为 0。

```
void  OS_MemInit(OS_ERR *p_err){
    #if OS_CFG_DBG_EN > 0u
    OSMemDbgListPtr = (OS_MEM    *)0;
    #endif
    OSMemQty = (OS_OBJ_QTY)0;
    *p_err = OS_ERR_NONE;
}
```

8.3.2　添加内存分区到调试列表

```
void  OS_MemDbgListAdd(OS_MEM   *p_mem)
```

参数说明：

p_mem：指向内存分区的指针。

使用说明：

该函数只有在 OS_CFG_DBG_EN＝1 时才能被系统创建，用于被 OSMemCreate()调用来添加内存分区到调试列表中。该函数也是 μC/OS-Ⅲ的内部函数，应用程序不能调用它。

```
void  OS_MemDbgListAdd(OS_MEM   *p_mem){
    p_mem->DbgPrevPtr = (OS_MEM *)0;
    if (OSMemDbgListPtr == (OS_MEM *)0) {
        p_mem->DbgNextPtr = (OS_MEM *)0;
    } else{
        p_mem->DbgNextPtr = OSMemDbgListPtr;
        OSMemDbgListPtr->DbgPrevPtr = p_mem;
    }
    //调试列表头指向此分区
    OSMemDbgListPtr = p_mem;
}
```

8.3.3　内存分区创建函数

```
void  OSMemCreate (OS_MEM          *p_mem,
 CPU_CHAR       *p_name,
```

```
void            * p_addr,
OS_MEM_QTY    n_blks,
OS_MEM_SIZE    blk_size,
OS_ERR          * p_err)
```

参数说明：

(1) p_mem：指向被分配的内存分区控制块的指针；

(2) p_name：指向代表内存分区名称的字符串；

(3) p_addr：指向内存分区的起始地址；

(4) n_blks：在内存分区中创建的内存块数量；

(5) blk_size：内存分区中每一个内存块的大小(以字节为单位)；

(6) p_err：指向包含错误码的变量指针。

其中，p_err 可能包含以下几种情况：

① OS_ERR_NONE：成功创建了内存分区；

② OS_ERR_ILLEGAL_CREATE_RUN_TIME：在调用了 OSSafetyCriticalStart()后调用了内存分区创建函数；

③ OS_ERR_MEM_INVALID_BLKS：没有为内存分区创建至少两个内存块；

④ OS_ERR_MEM_INVALID_P_ADDR：非法地址，内存地址取消或内存区和内存块边界没有对齐；

⑤ OS_ERR_MEM_INVALID_SIZE：用户指定了一个无效的内存块大小。内存块大小必须要大于一个指针变量且是指针变量的整数倍。

使用说明：

该函数创建并初始化一个用于动态内存分配的区域，该内存分区包含指定数目且大小确定的内存块。应用程序可以动态申请内存分区中的内存块，并且在使用完成后释放回内存分区中。该函数将内存分区的地址赋值给 p_mem，该指针也同时作为 OSMemGet()、OSMemPut()等相关函数调用的对象。内存分区创建函数具体流程如图 8.4 所示。

空闲内存块链表的创建过程如下：

```
p_link   =   (void * * )p_addr;                          (1)
p_blk    =   (CPU_INT08U * )p_addr;                      (2)
loops    =   n_blks - 1u;                                (3)
for (i = 0u; i < loops; i++)
{
    p_blk   +=   blk_size;                               (4)
     * p_link   =   (void   * )p_blk;                    (5)
    p_link   =   (void * * )(void * )p_blk;              (6)
}
 * p_link   =   (void * )0;                              (7)
```

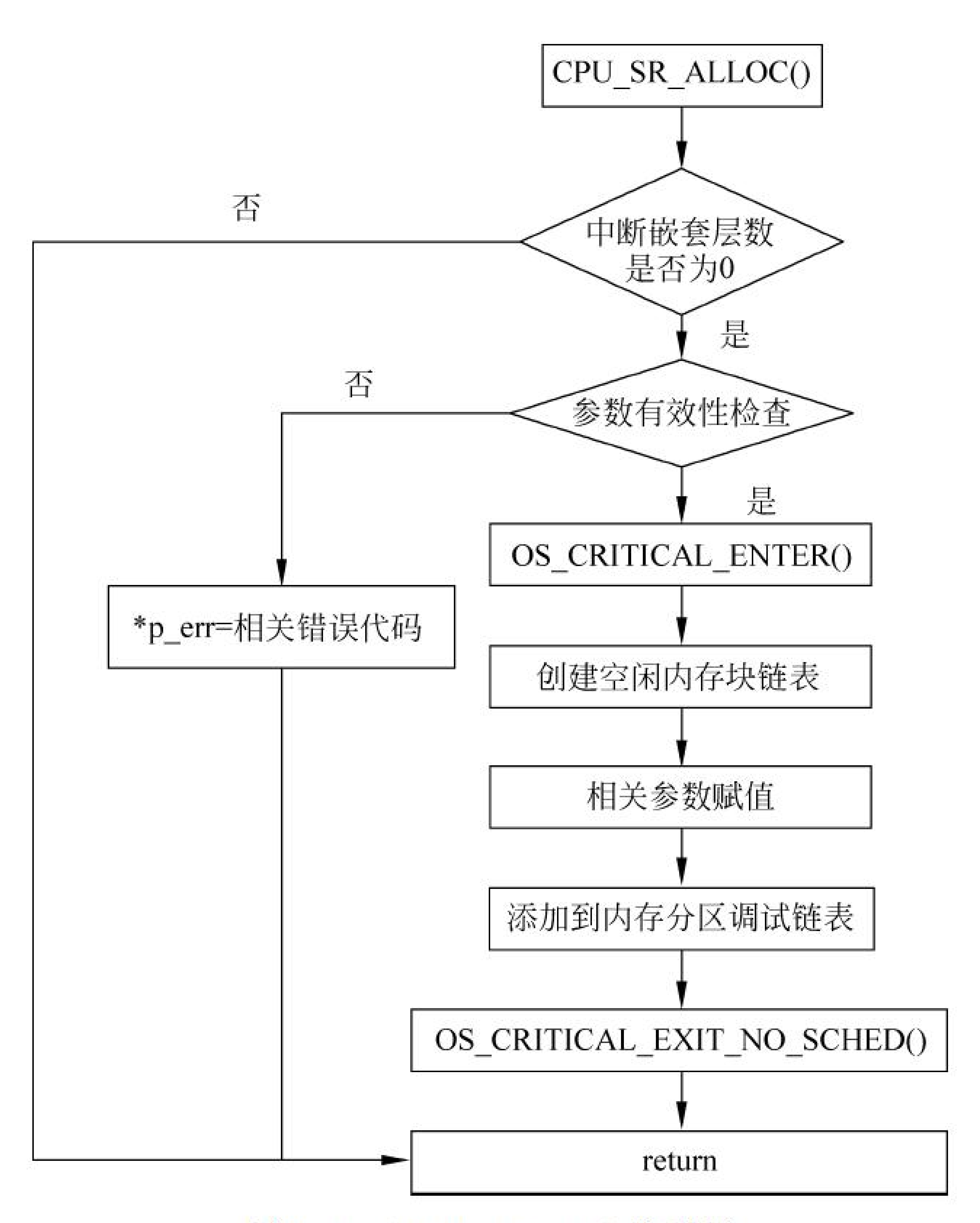


图 8.4 OSMemCreate()流程图

(1) 将内存分区的起始地址 p_addr 强制类型转换为二维指针赋值给 p_link，因此 p_link 作为一个二维指针指向内存分区起始地址。

(2) 将内存分区的起始地址 p_addr 赋值给 p_blk，并且进行 INT08U * 强制类型转化。因为在 μC/OS-Ⅲ的内存分区中，内存块是按照字节顺序分布的，便于第(4)步中内存块位置的计算。

(3) 待申请的内存块数量减 1，确定循环次数。在这个循环体中，通过不断计算下一个内存块的位置，完成空闲内存块链表的创建。

(4) 当前内存块位置指针 p_blk 加上一个内存块的大小，指向下一个内存块的位置。

(5) p_link 此时指向当前内存块，p_blk 指向下一个空闲内存块，在当前内存块中保存指向下一个内存块的指针。

(6) 将 p_blk 强制类型转换为二维指针赋值给 p_link，使其指向下一个内存块的位置。

(7) 最后一个内存块指向 NULL。

8.3.4 内存块获取函数

```
void    * OSMemGet(OS_MEM    * p_mem,
                   OS_ERR    * p_err)
```

参数说明：

(1) p_mem：指向内存分区控制块的指针，可以从 OSMemCreate()中获取；

(2) p_err：指向包含错误码变量的指针。

其中，p_err 可能包含以下几种情况：

① OS_ERR_NONE：成功获取到内存块；

② OS_ERR_MEM_INVALID_P_MEM：p_mem 参数无效，可能传递了一个空指针；

③ OS_ERR_MEM_NO_FREE_BLKS：内存分区中没有足够的空闲内存块。

使用说明：

该函数用于从内存分区中分配一个内存块。用户程序必须知道所建立的内存块的大小，并且在使用完内存块后必须释放它。可以多次调用 OSMemGet()函数。它的返回值是指向所分配内存块的指针，并且作为 OSMemPut()函数的参数。OSMemGet()函数具体流程图如图 8.5 所示。

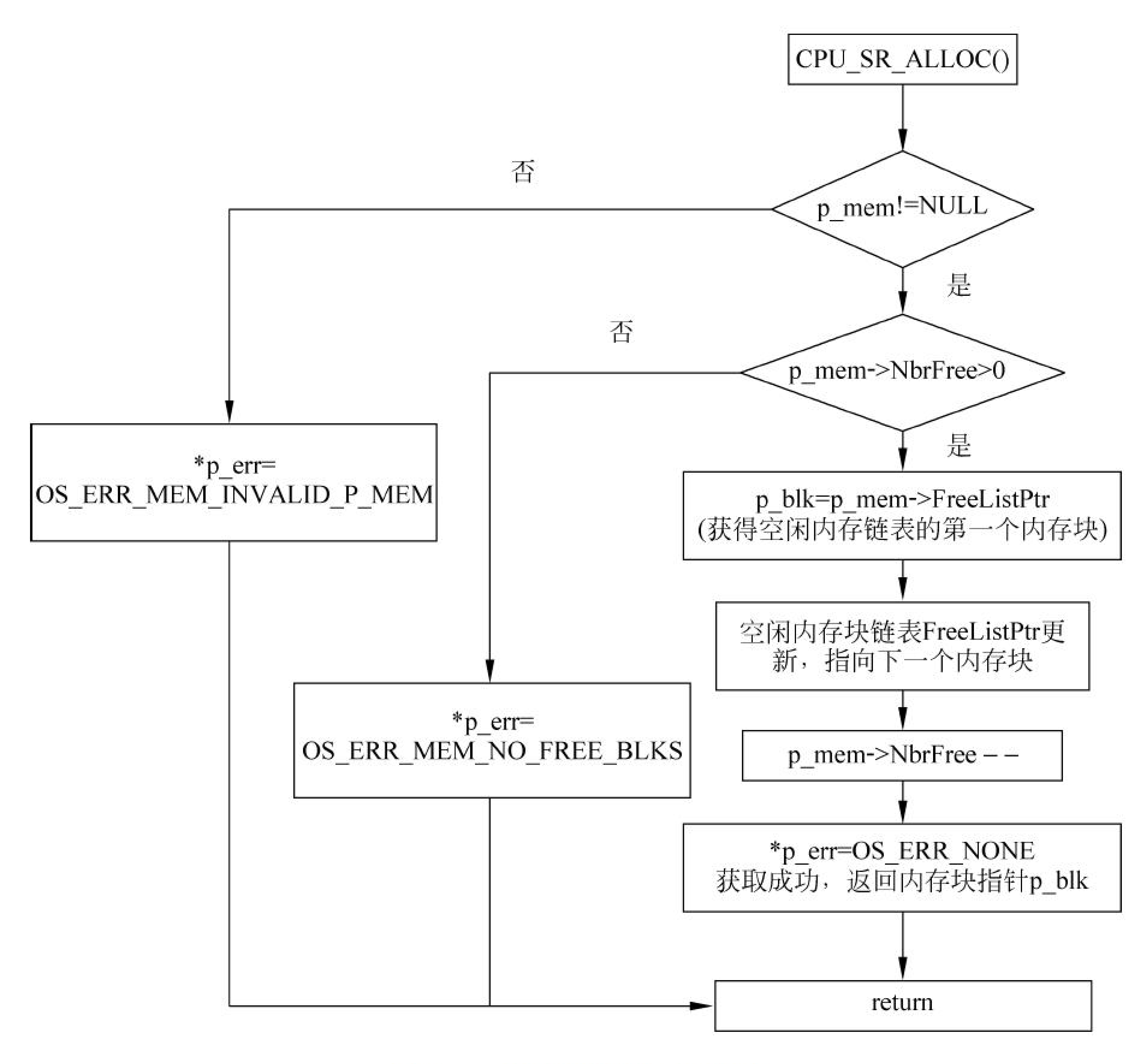


图 8.5 OSMemGet()流程图

OSMemGet()函数的关键代码如下：

```
p_blk = p_mem->FreeListPtr;                (1)
p_mem->FreeListPtr = *(void **)p_blk;      (2)
p_mem->NbrFree--;                          (3)
return (p_blk);                            (4)
```

（1）获取空闲内存块链表的指针。

（2）将空闲内存块链表的指针指向下一个空闲内存块。在 8.3.3 节中已经介绍过 OSMemCreate()函数创建过程中建立空闲内存块链表的过程。先将 p_blk 强制转换为二维指针，此时(void **)p_blk 中记录了下一个空闲内存块的地址，然后再对其进行取地址操作，得到下一个内存空闲块的地址。

（3）空闲内存块的数目减 1。

（4）返回申请的内存块的地址指针。

8.3.5　内存块释放函数

```
void  OSMemPut(OS_MEM   *p_mem,
               void     *p_blk,
               OS_ERR   *p_err)
```

参数说明：

（1）p_mem：指向内存分区控制块的指针，可以从 OSMemCreate()函数的返回值中得到；

（2）p_blk：指向将要被释放的内存块的指针；

（3）p_err：指向包含错误码变量的指针。

其中，p_err 可能包含以下几种情况：

① OS_ERR_NONE：内存块成功地插入到了内存分区中；

② OS_ERR_MEM_FULL：内存分区已满，无法插入内存块；

③ OS_ERR_MEM_INVALID_P_BLK：无效的内存块，比如传递了一个空指针给 p_blk；

④ OS_ERR_MEM_INVALID_P_MEM：无效的内存分区，比如传递了一个空指针给 p_mem。

使用说明：

该函数用于释放一个内存块，内存块必须释放回它原先所在的内存区域，否则会造成系统错误，如图 8.6 所示。

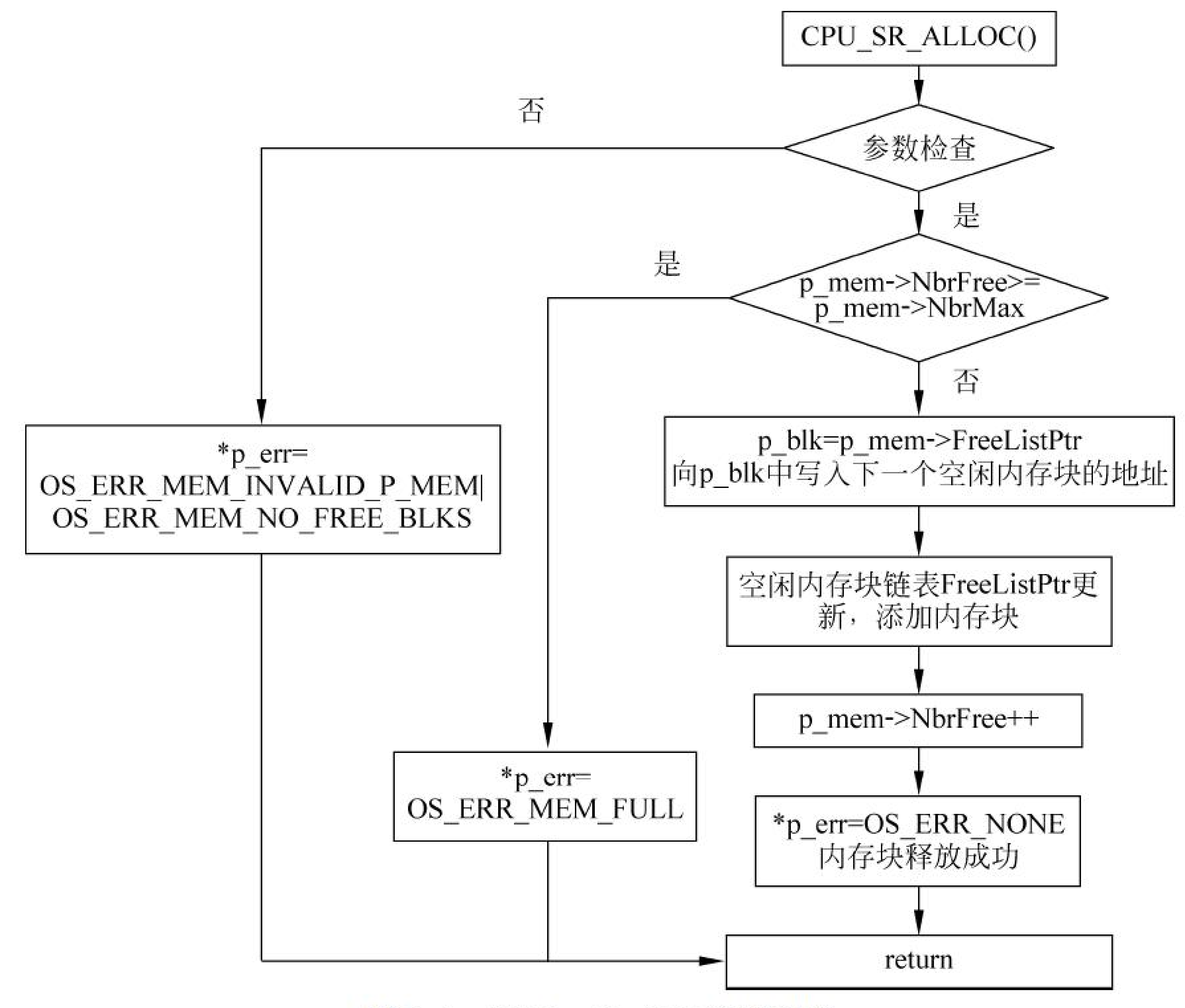


图 8.6 OSMemGet()函数流程图

8.4 内存管理应用开发

8.4.1 场景描述

设计一个有两个任务的应用程序，这两个任务分别是 Task0 和 Task1。在应用程序中创建一个动态内存分区，该分区有 6 个内存块，每个内存块大小为 20 个字节。任务 Task0 在申请了一个内存块后挂起自己，任务 Task1 将内存块全部赋值为 1，然后唤醒 Task0，Task0 将申请的内存释放。场景流程图如图 8.7 所示。

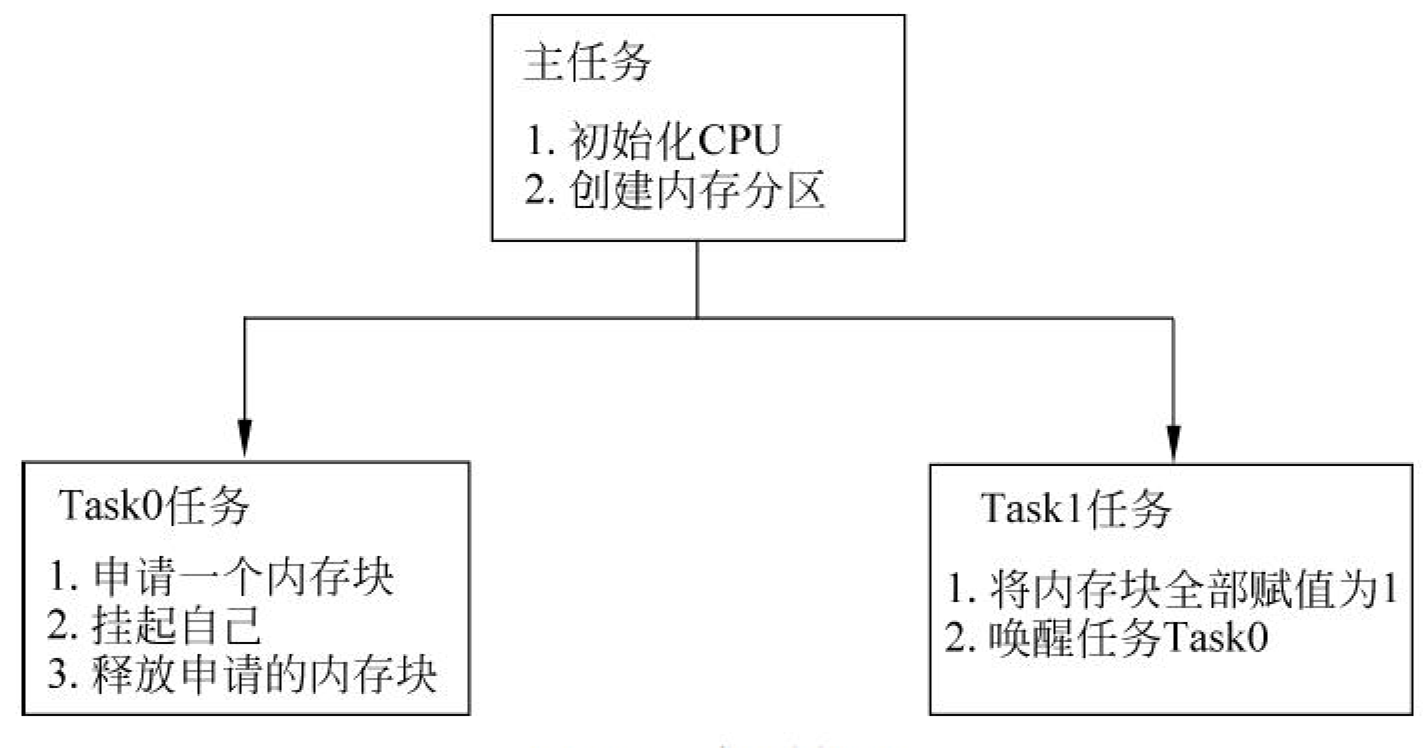


图 8.7 场景描述

8.4.2　设计环境

运行环境：软件平台 VS2013。

8.4.3　具体实现

app_cfg. h：

1. 定义各个任务的优先级

```
//定义任务的优先级
#define  APP_TASK_START_PRIO            4u
#define  TASK0_PRIO                     6u
#define  TASK1_PRIO                     7u
```

2. 定义任务栈的大小

```
#define  APP_TASK_START_STK_SIZE       1024u
#define  Task0StkLength                128u
#define  Task1StkLength                128u
```

app. c：

1. 定义各个任务 TCB、创建任务栈，定义内存管理需要的各个变量

```
//创建任务块
static   OS_TCB    MainTaskStartTCB;
static   OS_TCB    Task0TCB;
static   OS_TCB    Task1TCB;
//创建任务栈
static   CPU_STK    MainTaskStartStk[APP_TASK_START_STK_SIZE];
static   CPU_STK    WatcherStk[WatcherStkLength];
static   CPU_STK    Task0Stk[Task0StkLength];
static   CPU_STK    Task1Stk[Task1StkLength];
static   CPU_STK    Task2Stk[Task2StkLength];
//内存块指针
int * IntBlkPtr;
//内存分区控制块
OS_MEM INTERNAL_MEM;
//内存块数目
#define INTERNAL_MEM_NUM 6
//每一个内存块的大小
#define INTERNAL_MEMBLOCK_SIZE 20
//内存分区起始地址
CPU_INT08U
Internal_RamMemp[INTERNAL_MEM_NUM][INTERNAL_MEMBLOCK_SIZE];
```

2. 声明任务函数

```
static void MainTaskStart(void  * p_arg);
static void Task0(void  * p_arg);
static void Task1(void  * p_arg);
```

3. 在 main 函数中初始化系统、创建开始任务,之后启动多任务调用

```
int main(void){
    OS_ERR err;
    OSInit(&err);

    OSTaskCreate((OS_TCB * )&MainTaskStartTCB,
    (CPU_CHAR * )"Main Task Start",
    (OS_TASK_PTR)MainTaskStart,
    (void * ) 0,
    (OS_PRIO)APP_TASK_START_PRIO,
    (CPU_STK * )&MainTaskStartStk[0],
    (CPU_STK_SIZE)APP_TASK_START_STK_SIZE/10u,
    (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
    (OS_MSG_QTY)0u,
    (OS_TICK)0u,
    (void * )0,

    (OS_OPT)(OS_OPT_TASK_STK_CHK|OS_OPT_TASK_STK_CLR),(OS_ERR * )&err);
    OSStart(&err);
}
```

4. 在 MainTaskStart()中按照需求创建内存分区,同时创建 Task0 任务和 Task1 任务

```
static void MainTaskStart(void  * p_arg){
    OS_ERR err;
    (void)p_arg;
    BSP_Init();           //初始化 BSP
    CPU_Init();           //初始化 CPU
    #if OS_CFG_STAT_TASK_EN > 0u
    OSStatTaskCPUUsageInit(&err);
    #endif
    OSMemCreate((OS_MEM * )&INTERNAL_MEM,
        (CPU_CHAR * )"Internal Mem",
        (void * )&Internal_RamMemp[0][0],
        (OS_MEM_QTY)INTERNAL_MEM_NUM,
        (OS_MEM_SIZE)INTERNAL_MEMBLOCK_SIZE,
        (OS_ERR * )&err
        );
```

```
        APP_TRACE_DBG(("内存分区创建成功...\n\r"));

    OSTaskCreate()          //创建 Task0
    OSTaskCreate()          //创建 Task1
```

5. Task0 任务实现

使用 OSMemGet()内存块获取函数获取一块内存块,然后将自身任务挂起,被其他任务唤醒后输出内存块的值,然后释放申请的内存块。

```
static void Task0(void * p_arg){
    OS_ERR err;
    int count = 0;
    //获取内存块
    IntBlkPtr = OSMemGet(&INTERNAL_MEM, &err);
    if (err == OS_ERR_NONE){
        APP_TRACE_DBG(("Task0 成功申请了内存\n"));
        //挂起任务自己
        OSTaskSuspend((OS_TCB * )&Task0TCB, &err);
        for (; count < 5; count++){
            //输出内存块的值
            printf(" %d ", * (IntBlkPtr + count));
        }
        printf("\n");
        //释放内存块
        OSMemPut(&INTERNAL_MEM,IntBlkPtr,&err);
        APP_TRACE_DBG(("Task0 成功释放了内存\n"));
    }else{
        APP_TRACE_DBG(("Task0 申请内存失败\n"));
    }
}
```

6. Task1 任务实现

将任务 Task0 申请到的内存块全部赋值为 1,调用 OSTaskResume()唤醒 Task0。

```
static void Task1(void * p_arg){
    OS_ERR err;
    int count = 0;
    //内存块大小为 20 字节,所以有 5 个 int 值空间
    for (; count < 5; count++){
        * (IntBlkPtr + count) = 1;
    }
    APP_TRACE_DBG(("内存块全部赋值为 1\n"));
    OSTaskResume((OS_TCB * )&Task0TCB, &err);
}
```

8.4.4 实验结果

程序运行结果如图 8.8 所示。

```
内存分区创建成功...
Task0 成功申请了内存
内存块全部赋值为1
1 1 1 1 1
Task0 成功释放了内存
```

图 8.8 运行结果

习题

1. μC/OS-Ⅲ采用什么方法防止内存碎片的产生?
2. 直接使用 malloc/free 内存管理函数会导致什么问题? 为什么?
3. 内存控制块 os_mem 主要包含哪些变量?
4. 内存分区调试链表是如何将各个内存控制块连接起来的?
5. 内存初始化函数是如何初始化内存的?
6. 怎样将内存分区添加到调试列表?
7. μC/OS-Ⅲ是怎样创建内存分区的?
8. 内存控制块请求和释放的流程是什么?

第 9 章



文件系统 μC/FS

9.1 文件系统概述

μC/FS 是一款紧凑、可靠的高性能文件系统，它提供了设备文件目录访问和存储空间管理的功能。

μC/FS 文件系统有以下几个方面的特点。

1. 源代码开放

μC/FS 是用 ANSI-C 语言编写的，严格遵循编码规范，具有简洁性和良好的可读性的特点；遍布整个代码部分的注释阐明了代码的逻辑和全局的变量与函数；用标准 C 语言输入输出库的应用可以很容易地移植到 μC/FS 上。

2. FAT 支持

μC/FS 文件系统支持各种标准的 FAT 格式，包括 FAT12/FAT16/ FAT32 和长文件名，Unicode 文件名，最大支持 4GB 的文件和 8TB 的空间。文件配置表(File Allocation Table，FAT)，是一种由微软开发并拥有部分专利的文件系统，供 MS-DOS 使用，也是所有非 NT 核心的微软窗口使用的文件系统。考虑到电脑性能有限，所以 FAT 文件系统未被复杂化，因此支持几乎所有个人电脑的操作系统。这一特性使它成为理想的软盘和存储卡文件系统，也适用于不同操作系统中的数据交互。一个 FAT 文件系统包括四个不同的部分：保留扇区、FAT 区域、根目录区域和数据区域。FAT 有一个严重的缺点：当文件被删除并且在同一位置写入新数据时，它们的片段通常是分散的，这会降低读写速度。

3. OS 支持

μC/FS 可以容易地整合到任何操作系统上，因此可以在多线程环境下进行文件操作。

4. 设备驱动

在大多数媒质上都有可用的设备驱动，例如：SD/MMC cards、NAND flash、NOR flash 等。驱动程序有着简单、清晰分层的结构(一些基本的读写函数)，它能十分容易地移植到硬件，甚至新的媒质上。

5. 设备和容积

多种多样的设备可以同时接入 μC/FS 文件系统,即便它们属于同一类型。一个设备的空间可以被分为多个区域,一块空间可以跨越多个设备。

6. 可伸缩

μC/FS 的内存占用可以根据特征需求和运行时的参数检测级别进行调整。对于内存有限制的应用,像高速缓存和读写缓冲区的特征可以被取消。

7. 可移植

μC/FS 为资源受约束的嵌入式应用而设计。虽然它也可以在 8 位和 16 位的处理器上工作,但最好还是工作在 32 位或 64 位的处理器上。

8. RTOS

μC/FS 并没有专门为 RTOS 内核而设计。如果使用的是实时操作系统,则需要添加一个简单的传输层(由一些信号量组成),防止多个任务同时对核访问。

μC/FS 的结构如图 9.1 所示。

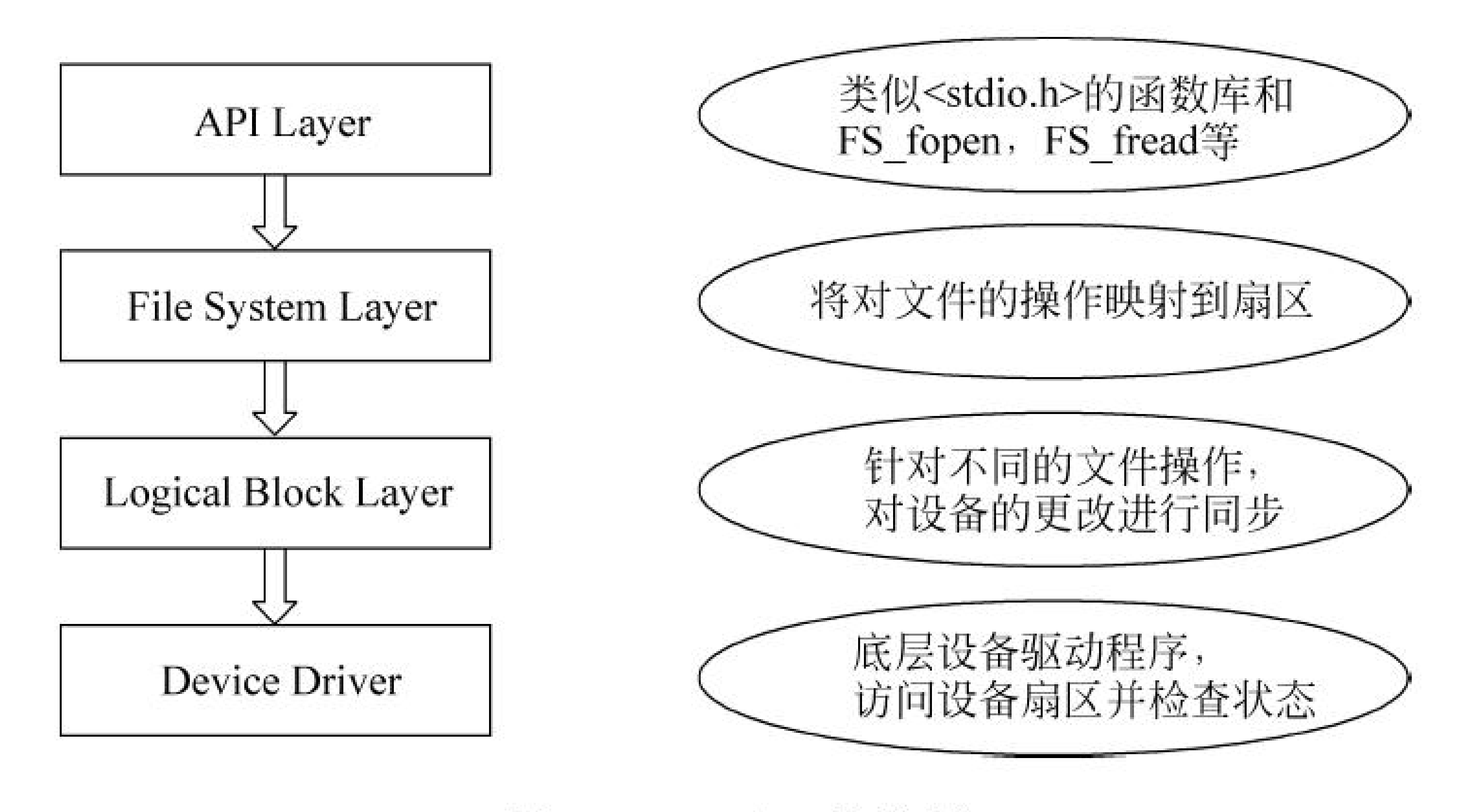


图 9.1 μC/FS 结构图

9. API 层

API 层是 μC/FS 提供给用户使用的接口(fs_api.h),位于文件系统和用户应用之间。它包括一系列面向 ANSI C 的功能函数,如 FS_FOpen,FS_FWrite 等,API 层将各种调用传输到 file system layer(文件系统层)。目前对 μC/FS(文件管理实现机制)而言只有一个 FAT 文件系统层被使用,但 API 层可以同时处理不同的文件系统层,因此 μC/FS 可以同时支持 FAT 和其他的文件系统。

10. File System Layer 文件系统层

文件系统层将文件操作转换为逻辑块操作,具体的文件系统调用逻辑块层函数并指定相应的设备驱动。

uC_FS\FSL\fat\下为 FAT 文件系统的各个文件。

11. Logical Block Layer 逻辑块层

逻辑块层的主要目的是同步访问设备驱动与文件系统层的简易接口,逻辑块层调用设

备驱动来实现设备的块操作。

12. Device Driver Layer 设备驱动层

设备驱动是一些访问硬件的底层例行操作，μC/FS 的设备驱动架构很简单，可以被容易地整合到硬件上。

9.2 机制方法

用户可以通过调用 μC/OS 文件系统提供的 API 函数新建、打开、读写、删除文件，但真正对设备存储空间进行操作的却是内核函数，内核函数对设备进行格式化（初始化），给文件分配簇来存储文件内容，一个簇又由多个扇区组成，每个扇区存储文件的一部分。

用 API 函数对文件进行操作时，需要用参数传递操作模式，是读、写等不同的模式，API 函数并不直接调用内核函数进行设备操作，设备操作的内核函数封装在数据结构 FS_fsl_type 中，类似于 Java 中的接口继承，API 函数将参数传递给 FS_fsl_type，通过此数据结构来调用内核函数进行设备操作，如图 9.2 所示。

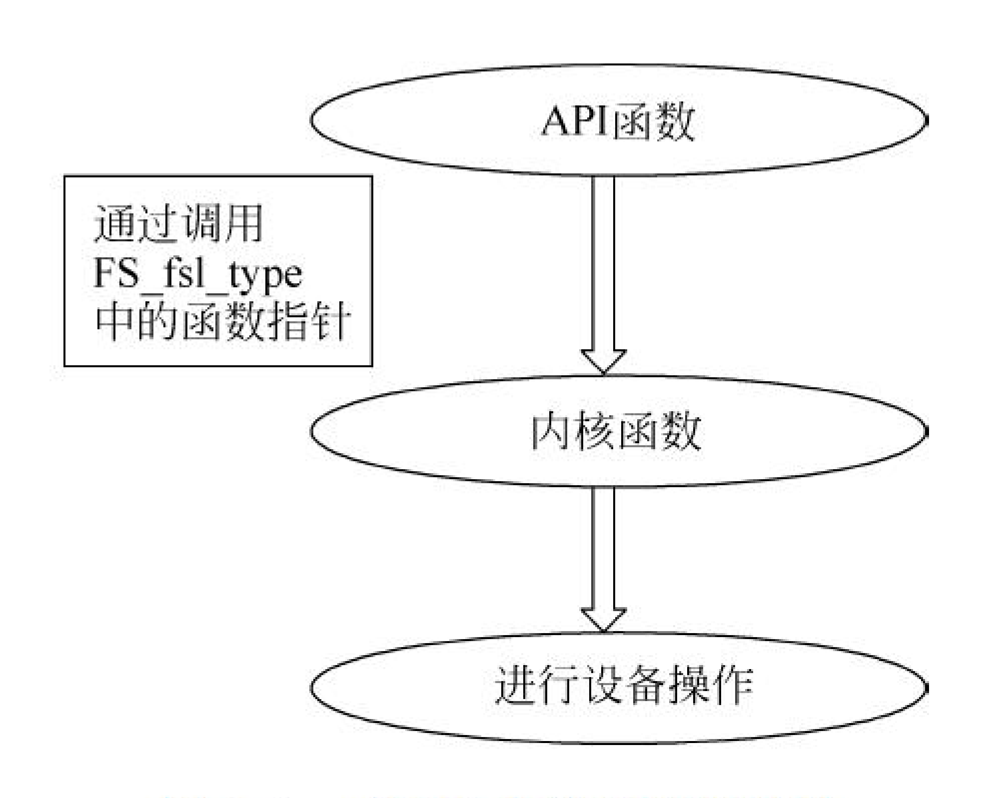


图 9.2 μC/FS 内部函数调用图

9.3 关键数据结构

μC/FS 定义了一些数据结构来管理文件系统。根据操作对象不同可将数据结构分为定义文件的数据结构和定义文件夹的数据结构（详见 fs_api.h）。

9.3.1 文件及文件操作的数据结构

FS_FILE 定义了一个文件的具体信息，将一个文件当作一个操作对象来管理；FS_fsl_type 封装了文件操作的标识，用来判断对文件进行了哪些操作，将文件操作当作一个对象来管理；FS_devinfo_type 进一步封装了文件操作的具体信息，如操作的存储器，操作类型，以及驱动类型，声明 FS_devinfo_type 类型的数组来存储操作，同时按数组顺序来执行操

作。在这里 FS_devinfo_type 相当于文件控制块，它以数组的方式组织。它们之间的关系如图 9.3 所示。

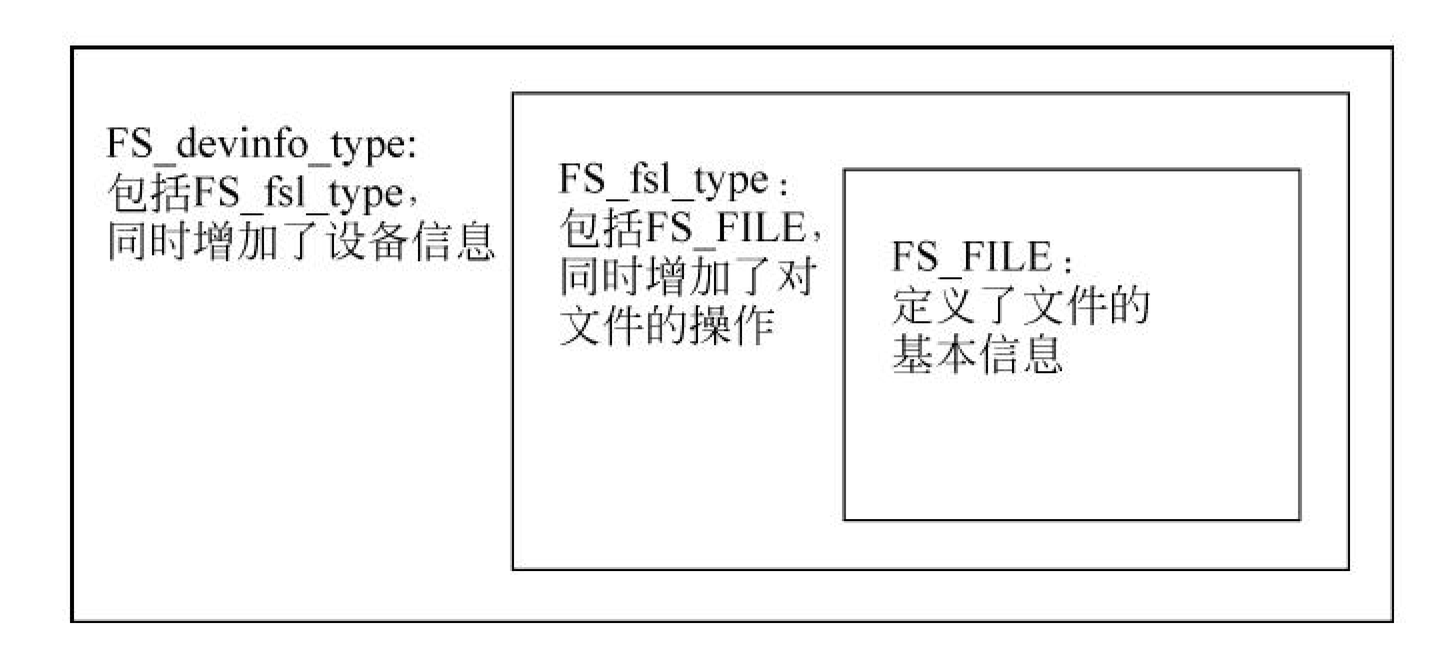


图 9.3 文件系统数据结构组织图

(1) FS_FILE，定义了一个文件的具体信息，其定义如下：

```
typedef struct {
    FS_u32 fileid_lo;            //文件唯一 ID 标识符(lo),FS_u32:32bit unsigned long
    FS_u32 fileid_hi;            //文件唯一 ID 标识符(hi)
    FS_u32 fileid_ex;            //文件唯一 ID 标识符(ex)
    FS_i32 EOFClust;
    FS_u32 CurClust;
    FS_i32 filepos;              //在文件中指针(current)的位置 32bit signed long
    FS_i32 size;                 //文件大小
    int dev_index;               //在 in _FS_devinfo[]中的索引
    FS_i16 error;                //错误编码
    unsigned char inuse;
    unsigned char mode_r;        //mode READ 标识为读模式
    unsigned char mode_w;        //mode WRITE 标识为写模式
    unsigned char mode_a;        //mode APPEND 标识为附加模式
    unsigned char mode_c;        //mode CREATE 标识为新建模式
    unsigned char mode_b;        //mode BINARY 标识为二进制模式
} FS_FILE;
```

FS_FILE 定义了一个文件的具体信息，将一个文件当作一个操作对象来管理。

(2) FS_fsl_type，它用来存储文件操作，其定义如下：

```
typedef struct {
    FS_FARCHARPTR          name;
    //打开一个文件，返回值为文件的指针

    FS_FILE *      ( * fsl_fopen)(const char * pFileName,
                    const char * pMode, FS_FILE * pFile);
```

```
    void( * fsl_fclose)(FS_FILE * pFile);                          //关闭文件
        //读文件
    FS_size_t( * fsl_fread)(void * pData, FS_size_t Size, FS_size_t N, FS_FILE * pFile);
    FS_size_t( * fsl_fwrite)(const void * pData, FS_size_t Size,
                              FS_size_t N, FS_FILE * pFile);   //写文件
    long( * fsl_ftell)(FS_FILE * pFile);
    int( * fsl_fseek)(FS_FILE * pFile, long int Offset, int Whence);  //查找文件
    int( * fsl_ioctl)(int Idx, FS_u32 Id, FS_i32 Cmd, FS_i32 Aux, void * pBuffer);
#if FS_POSIX_DIR_SUPPORT
        //以下是对文件夹的操作
        FS_DIR * ( * fsl_opendir)(const char * pDirName, FS_DIR * pDir);//打开文件夹
        int( * fsl_closedir)(FS_DIR * pDir); //关闭文件夹
        struct FS_DIRENT * ( * fsl_readdir)(FS_DIR * pDir);         //读文件夹
        //返回文件夹开始第一个文件(夹)
        void( * fsl_rewinddir)(FS_DIR * pDir);
        int( * fsl_mkdir)(const char * pDirName, int DevIndex, char Aux);//新建文件夹
        int( * fsl_rmdir)(const char * pDirName, int DevIndex, char Aux);//删除文件夹
#endif
} FS_fsl_type;
```

FS_fsl_type 封装了文件操作的标识，用来判断对文件进行了哪些操作，将文件操作当作一个对象来管理。

(3) FS_devinfo_type，其定义如下：

```
typedef struct {
    const char              * const devname;        //用来区分硬件设备
    const FS_fsl_type       * const fs_ptr;         //文件类型
    const FS_device_type    * const devdriver;      //设备类型
#if FS_USE_LB_READCACHE
    FS_LB_CACHE             * const pDevCacheInfo;
#endif
    const void              * const data;
} FS_devinfo_type;
```

FS_devinfo_type 进一步封装了文件操作的具体信息，如操作的存储器，操作类型，以及驱动类型。

FS_devinfo_type 声明 FS_devinfo_type 类型的数组来存储操作，同时按数组顺序来执行操作。

```
const FS_devinfo_type * const FS_pDevInfo = _FS_devinfo;
const FS_devinfo_type _FS_devinfo[] = { FS_DEVINFO };
```

9.3.2 文件夹数据结构

文件夹数据结构如图 9.4 所示。



图 9.4 文件系统文件夹数据结构

(1) FS_DIRENT：FS_DIRENT 定义了文件夹的一些基本信息。

```
#define FS_ino_t  int
struct FS_DIRENT {
    FS_ino_t  d_ino;
    char      d_name[FS_DIRNAME_MAX];          //文件夹名称
    char      FAT_DirAttr;                     //FAT 目录属性
};
```

(2) FS_DIR：FS_DIR"继承"了 FS_DIRENT，同时加入了更多的信息。

```
typedef struct {
    struct FS_DIRENT  dirent;                  //当前文件夹的入口
    FS_u32 dirid_lo;                           //文件夹唯一标识符(lo)
    FS_u32 dirid_hi;                           //文件夹唯一标识符(hi)
    FS_u32 dirid_ex;                           //文件夹唯一标识符(ex)
    FS_i32 dirpos;                             //文件夹内部指针
    FS_i32 size;                               //文件夹大小
    int dev_index;                             //in _FS_devinfo[]索引
    FS_i16 error;                              //错误代码
    unsigned char inuse;
} FS_DIR;
```

9.3.3 其他的一些变量及数据结构

(1) FS_ramdevice_driver：这个 FS_device_type 数组定义了一些存储器的类型值，例如：RAMDISK device，简称 ram。新建文件(夹)时可以用参数指定所选的存储器类型。默认为 RAMDISK device。

```
const FS_device_type FS_ramdevice_driver = {
    "RAMDISK device",
    _FS_RAM_DevStatus,
    _FS_RAM_DevRead,
```

```
    _FS_RAM_DevWrite,
    _FS_RAM_DevIoCtl
};
```

（2）FS_device_type：定义了 device 的名称和对应的操作。

```
typedef struct {
    FS_FARCHARPTR     name;
    int       ( * dev_status)(FS_u32 id);
    int       ( * dev_read)(FS_u32 id, FS_u32 block, void * buffer);
    int       ( * dev_write)(FS_u32 id, FS_u32 block, void * buffer);
    int       ( * dev_ioctl)(FS_u32 id, FS_i32 cmd, FS_i32 aux, void * buffer);
} FS_device_type;
```

（3）信号量：μC/OS 在进行文件操作的时候，为了支持多线程，同时定义了一些信号量，以此来控制进程间文件操作的同步，并保证数据安全性和完整性。

```
static HANDLE _FS_fh_sema = NULL;
static HANDLE _FS_fop_sema = NULL;
static HANDLE _FS_mem_sema = NULL;
static HANDLE _FS_dop_sema = NULL;

#if FS_POSIX_DIR_SUPPORT     //以下为不同进程对文件夹操作而定义了一些信号量
static HANDLE _FS_dirh_sema = NULL;
static HANDLE _FS_dirop_sema = NULL;
#endif
```

（4）文件操作模式相应的数据结构。

```
typedef struct {
 FS_FARCHARPTR mode;
 unsigned char mode_r;                              //读模式
 unsigned char mode_w;                              //写模式
 unsigned char mode_a;                              //索引模式
 unsigned char mode_c;                              //新建模式
 unsigned char mode_b;                              //二进制模式
} _FS_mode_type;
```

_FS_mode_type 数组定义了所有基本的操作模式，对文件进行操作时，将相应的参数与数组中的元素进行比较从而对 FS_FILE 中的数据项进行赋值。

```
static const _FS_mode_type _FS_valid_modes[] = {
/*    READ   WRITEAPPENDCREATE   BINARY */
  { "r",      1,     0,     0,     0,     0},
  { "w",      0,     1,     0,     1,     0},
  { "a",      0,     1,     1,     1,     0},
```

```
    { "rb",    1,    0,    0,    0,    1},
    { "wb",    0,    1,    0,    1,    1},
    { "ab",    0,    1,    1,    1,    1},
    { "r+",    1,    1,    0,    0,    0},
    { "w+",    1,    1,    0,    1,    0},
    { "a+",    1,    1,    1,    1,    0},
    { "r+b",   1,    1,    0,    0,    1},
    { "rb+",   1,    1,    0,    0,    1},
    {"w+b",    1,    1,    0,    1,    1},
    {"wb+",    1,    1,    0,    1,    1},
    { "a+b",   1,    1,    1,    1,    1},
    { "ab+",   1,    1,    1,    1,    1},
};
```

9.4 内核函数

μC/FS 文件系统定义了很多用户接口来实现文件系统的管理，但真正对文件系统进行操作的函数是一些内核函数。图 9.5 中列出了主要的内核函数。

- uC/FS
  - fat_dir.c
    - _FS_fat_create_directory(int, FS_u32, const char *, FS_u32 , FS_u32)
    - FS_fat_opendir(const char *, FS_DIR *)
    - FS_fat_closedir(FS_DIR *)
    - FS_fat_readdir(FS_DIR *)
    - FS_fat_MkRmDir(const char *, int, char)
  - fat_in.c
    - FS_fat_fread(void *, FS_size_t, FS_size_t, FS_FILE *)
  - fat_out.c
    - _FS_fat_write_dentry(int, FS_u32, FS_u32, FS__fat_dentry_type *, FS_u32 , char *)
    - _FS_fat_read_dentry(int, FS_u32, FS_u32, FS_u32, FS__fat_dentry_type *, FS_u32 *, char *)
    - FS_fat_fwrite(const void *, FS_size_t, FS_size_t, FS_FILE *)
    - FS_fat_fclose(FS_FILE *)
  - fat_open.c
    - _FS_fat_find_file(int, FS_u32, const char *,FS__fat_dentry_type *,FS_u32, FS_u32)
    - _FS_fat_IncDir(int, FS_u32, FS_u32, FS_u32 *)
    - _FS_fat_create_file(int, FS_u32, const char *,FS_u32, FS_u32)
    - FS_fat_DeleteFileOrDir(int, FS_u32, const char *,FS_u32, FS_u32, char)
    - FS_fat_make_realname(char *, const char *)
    - FS_fat_find_dir(int, FS_u32, char *, FS_u32,FS_u32)
    - FS_fat_dir_realsec(int, FS_u32, FS_u32, FS_u32)
    - FS_fat_findpath(int, const char *, FS_FARCHARPTR *,FS_u32 *, FS_u32 *)
    - FS_fat_fopen(const char *, const char *, FS_FILE *)
    - FS_fat_dir_size(int, FS_u32, FS_u32)

图 9.5 主要的内核函数图

9.4.1 _FS_fat_find_file()

_FS_fat_find_file()在给定设备目录下寻找名为“* pFileName”的文件，并将内容复制到“* pDirEntry”下，流程图如图 9.6 所示。

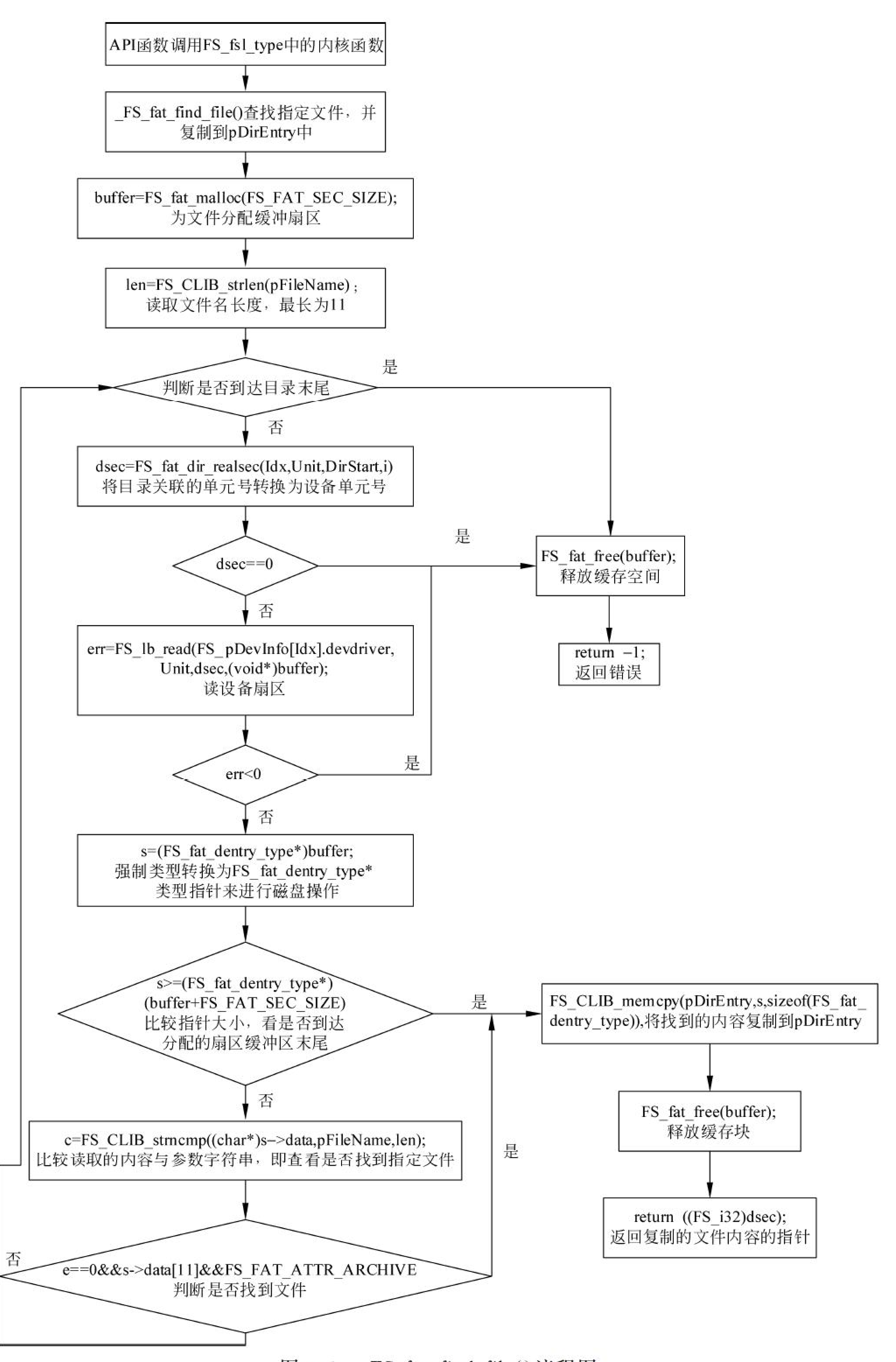


图 9.6　_FS_fat_find_file()流程图

9.4.2 _FS_fat_create_file()

_FS_fat_create_file()在给定设备目录下新建一个文件,不会检查是否重名,检查重名需要用户在 API 层进行,程序流程图如图 9.7 所示。

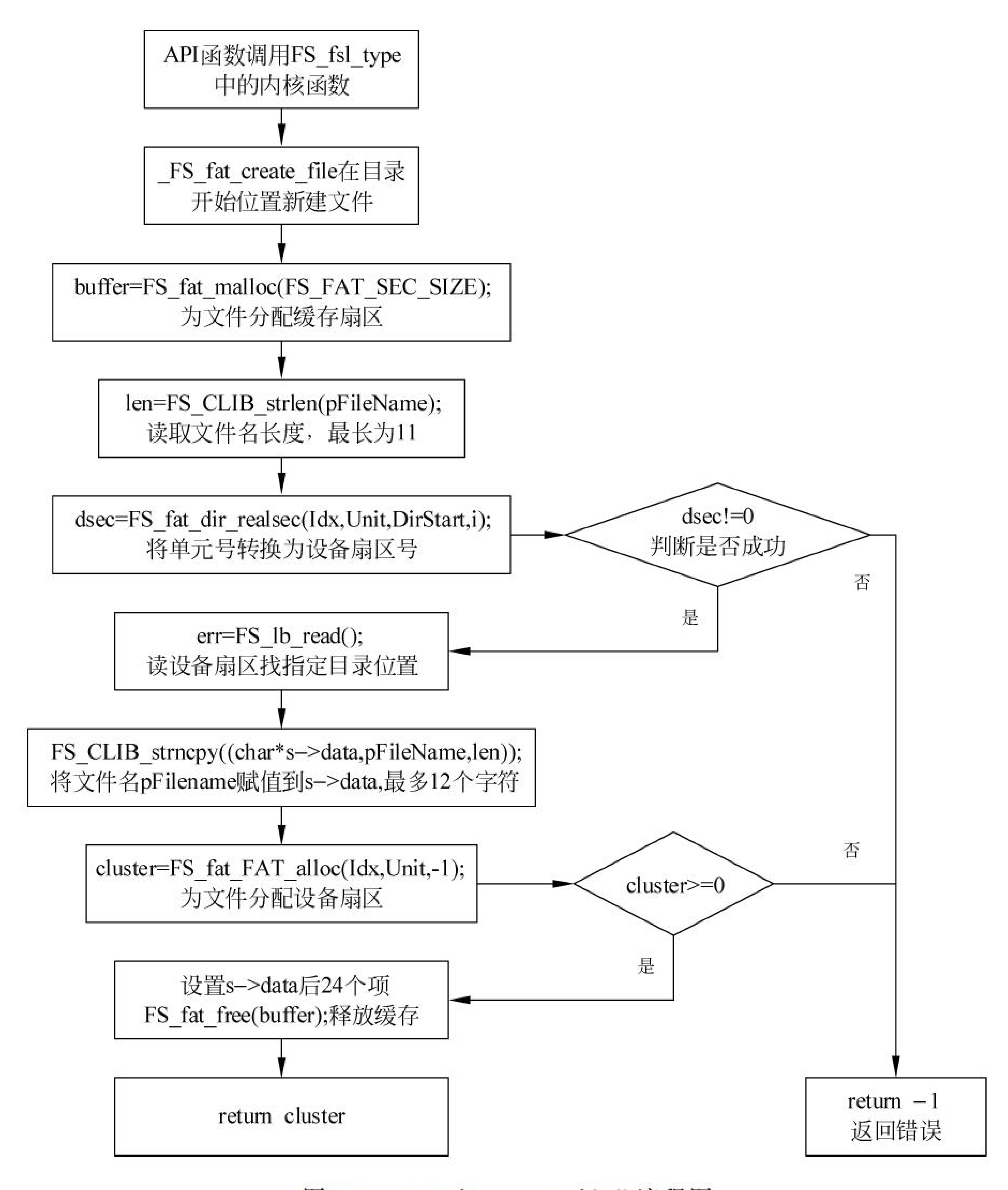


图 9.7 _FS_fat_create_file()流程图

9.5 应用函数介绍

1. 应用函数 API

μC/FS 定义了大量 API 函数,对文件系统进行管理操作。API 函数根据操作对象的不

同划分如下：

（1）对文件夹的操作，如新建、删除文件夹，遍历文件夹，输出文件夹下的内容。

（2）对文件的操作，如新建、删除文件，对文件进行读、写操作。

μC/FS 是通过名称来区分文件夹与文件的，有后缀的名称识别为文件，例如“a. txt”；没有后缀的名称识别为文件夹，例如“my”。

2. 操作简要介绍

（1）对文件夹的主要操作如下：

① FS_MkDir()：新建目录文件夹，返回非－1 表示成功；

② FS_OpenDir()：打开目录文件夹，成功返回文件夹指针；

③ FS_ReadDir()：读取目录下的文件/文件夹，成功返回目录入口结构体指针；

④ FS_RewindDir()：返回目录入口；

⑤ FS_RmDir()：删除目录，返回非－1 表示成功；

⑥ FS_CloseDir()：关闭目录，返回非－1 表示成功。

（2）对文件的主要操作如下：

① FS_FOpen()：按照参数中的模式打开文件并更新到文件指针的模式，若没有，则新建该文件，成功返回文件指针；

② FS_FClose()：关闭文件；

③ FS_FRead()：按照读文件的方式打开文件，成功返回文件指针；

④ FS_FWrite()：按照写文件的方式打开文件，成功返回文件大小；

⑤ FS_FSeek()：定位文件内部数据指针，成功返回 0；

⑥ FS_FTell()：寻找文件内部数据指针，成功返回指针位置；

⑦ FS_Remove()：删除文件，返回非－1 表示成功。

9.5.1 FS_Fopen()文件打开函数

FS_Fopen()按照参数中的模式打开文件并更新到文件指针的模式，若没有，则新建该文件，成功则返回文件指针。

```
FS_FILE * FS_FOpen(const char * pFileName, const char * pMode) {
    FS_FARCHARPTR s;
    FS_FILE * handle;
    unsigned int i;
    int idx;
    int j;
    int  c;
/ * Find correct FSL(device:unit:name) * /
    //将 pFilename 中的文件名分离出来,保存到 s 中
    idx = FS_find_fsl(pFileName, &s);
    if (idx < 0) {
    return 0;                                   //设备没有找到
    }
```

```
if (FS_pDevInfo[idx].fs_ptr->fsl_fopen) {
    //在_FS_filehandle 中寻找下一个条目
    FS_X_OS_LockFileHandle();              //申请信号量,对文件进行操作
    i = 0;
    while (1) {
        if (i >= _FS_maxopen) {
        break;                             //没有空闲的条目
        }
        if (!_FS_filehandle[i].inuse) {
        break;                             //查找到未使用的条目
        }
        i++;
    }
    if (i < _FS_maxopen) {
        //检查有效的模式字符串并且在文件处理中设置 flags
        j = 0;
        while (1) {
        if (j >= FS_VALID_MODE_NUM) {
            break;
        }
        c = FS_CLIB_strcmp(pMode, _FS_valid_modes[j].mode);
        if (c == 0) {
            break;
        }
        j++;
            }
if (j < FS_VALID_MODE_NUM) {
                //根据对文件操作的方式设置模式标志位
    _FS_filehandle[i].mode_r = _FS_valid_modes[j].mode_r;
    _FS_filehandle[i].mode_w = _FS_valid_modes[j].mode_w;
        _FS_filehandle[i].mode_a = _FS_valid_modes[j].mode_a;
        _FS_filehandle[i].mode_c = _FS_valid_modes[j].mode_c;
        _FS_filehandle[i].mode_b = _FS_valid_modes[j].mode_b;
            }else {
        FS_X_OS_UnlockFileHandle();        //释放信号量
    return 0;
        }
    _FS_filehandle[i].dev_index = idx;
    handle = (FS_pDevInfo[idx].fs_ptr->fsl_fopen)
                                (s, pMode, &_FS_filehandle[i]);
    FS_X_OS_UnlockFileHandle();            //释放信号量
    return handle;
    }
    FS_X_OS_UnlockFileHandle();            //释放信号量
    }
    return 0;
}
```

9.5.2 FS_FWrite()文件写入函数

FS_FWrite()按照写文件的方式打开文件,成功返回文件大小。

```
FS_size_t FS_FWrite(const void * pData, FS_size_t Size, FS_size_t N, FS_FILE * pFile) {
    FS_size_t i;
    if (!pFile) {
        return 0;                          //没有指向 FS_FILE 结构体的指针
    }
    FS_X_OS_LockFileOp(pFile);             //申请写文件操作信号量
    if (!pFile->mode_w) {
    //写模式打开文件
    pFile->error = FS_ERR_READONLY;
    FS_X_OS_UnlockFileOp(pFile);           //释放写文件操作信号量
    return 0;
    }
    i = 0;
    if (pFile->dev_index >= 0) {
    if (FS_pDevInfo[pFile->dev_index].fs_ptr->fsl_fwrite) {
        //执行 FSL 函数
        i = (FS_pDevInfo[pFile->dev_index].fs_ptr->fsl_fwrite)
                                              (pData, Size, N, pFile);
    }
    }
    FS_X_OS_UnlockFileOp(pFile);
    return i;
}
```

9.5.3 FS_FClose()文件关闭函数

```
void FS_FClose(FS_FILE * pFile) {
    if (!pFile) {
    return;                            //没有指向 FS_FILE 指针的结构
    }
    FS_X_OS_LockFileHandle();          //申请文件操作信号量
    if (!pFile->inuse) {
    FS_X_OS_UnlockFileHandle();        //这个 FS_FILE 结构体没有被使用
    return;
    }
    if (pFile->dev_index >= 0) {
    if (FS_pDevInfo[pFile->dev_index].fs_ptr->fsl_fclose) {
```

```
            (FS_pDevInfo[pFile->dev_index].fs_ptr->fsl_fclose)(pFile);
        }
        }
        FS_X_OS_UnlockFileHandle();                //释放文件操作信号量
    }
```

9.6 应用示例

9.6.1 场景描述

文件系统最主要的操作就是新建文件(夹)、写文件、读文件(夹)、删除文件(夹)。本示例将实现上述文件和文件夹的操作。

9.6.2 设计过程

```
//在设备默认目录下新建文件
_write_file("default.txt",dev_default_msg);
//在设备指定目录下新建文件夹
a = FS_MkDir("ram:\\my");
a = FS_MkDir("ram:\\my\\a");
a = FS_MkDir("ram:\\my\\b");
//在指定目录下新建文件
_write_file("ram:\\my\\a\\1.txt",dev_ram_msg);
_write_file("ram:\\ram.txt", dev_ram_msg);
//读文件
_dump_file("default.txt");
_dump_file("ram:\\ram.txt");
//读文件夹
_show_directory("");
_show_directory("ram:");
_show_directory("ram:\\my");
_show_directory("ram:\\my\\a");
//删除文件
a = FS_Remove("ram:\\my\\a\\1.txt");
_show_directory("ram:\\my\\a");
a = FS_RmDir("ram:\\my\\a");
_show_directory("ram:\\my");
```

9.6.3 测试

运行结果如图 9.8 所示。

```
succeed
succeed
succeed
This text was written on your default device.
This text was written on your RAM disk.
Directory of
DEFAULT .TXT
MY      .
RAM     .TXT
Directory of ram:
DEFAULT .TXT
MY      .
RAM     .TXT
Directory of ram:\my
.       .
..      .
A       .
B       .
Directory of ram:\my\a
.       .
..      .
1       .TXT
Directory of ram:\my\a
.       .
..      .
Directory of ram:\my
.       .
..      .
B       .
Disk information of
total clusters     : 25
available clusters : 21
sectors/cluster    : 1
bytes per sector   : 512
Disk information of ram:
total clusters     : 25
available clusters : 21
sectors/cluster    : 1
bytes per sector   : 512
请按任意键继续. . .
```

图 9.8　运行结果

习题

1. μC/FS 文件系统是一种什么文件系统？它有什么特点？
2. 请简述 μC/FS 文件系统结构。
3. μC/FS 文件系统内部调用结构是什么？
4. μC/FS 文件系统文件和文件操作的数据结构是什么？
5. μC/FS 文件系统文件夹数据结构是什么？
6. μC/FS 文件系统是如何创建一个文件的？
7. μC/FS 文件系统在打开和关闭文件时做了哪些工作？
8. μC/FS 文件系统是如何读写文件的？

第 10 章



μC/OS-Ⅲ移植

10.1　移植机制

当今时代，硬件的发展越来越多样化，微处理器和控制器的类型也越来越多，Intel、AMD、ARM 等处理器厂商也在嵌入式芯片领域激烈竞争，这些对嵌入式操作系统提出了空前的性能要求。操作系统要想提高生存能力，必须要能够在不同的处理器上运行，因此操作系统的可移植性变得至关重要。μC/OS-Ⅲ实时操作系统使用 C 语言和汇编语言编写，具有高度的可移植性，因此可以将 μC/OS-Ⅲ操作系统移植到不同类型的处理器上运行。

在移植过程中，和处理器相关的部分，仍然需要使用 C 语言和汇编语言编写，与操作寄存器相关的代码需要使用汇编语言编写，如果编译器支持内嵌汇编语言，则也可以采用 C 语言内嵌汇编语言的形式编写。由于 μC/OS-Ⅲ操作系统使用 C 语言和汇编语言编写，因此移植过程相对而言是比较容易的。移植代码编写成功后，需要使用汇编器和 C 语言编译器生成可重入代码，再由链接器将可重入代码链接生成可执行代码，最后将可执行代码烧写到目标板上成功执行，完成移植。

虽然 μC/OS-Ⅲ操作系统可以在大多数处理器上成功移植，但是要使 μC/OS-Ⅲ能够在处理器上正常运行，处理器至少需要满足以下条件：

(1) 处理器必须要有相应的汇编器、C 语言编译器和连接器，才能够生成处理器可以运行的可重入代码。一般来说，每款处理器都有相应的工具链来支持生成可重入代码。

(2) 处理器必须能够支持中断处理和硬件定时器，且能够产生时钟中断。现代处理器基本上都能够支持中断处理，并且通过晶振给处理器提供稳定的硬件时钟。

(3) 处理器必须要有足够的内存空间来存储 μC/OS-Ⅲ代码、数据变量和任务运行需要的任务栈空间，并且要有一套完整的支持内存读取操作和寄存器读取操作的指令。

μC/OS-Ⅲ的文件体系结构如图 10.1 所示，用户可以修改 μC/OS-Ⅲ中与移植和操作系统相关的部分来完成移植操作。

①配置文件

cpu_cfg.h 定义CPU相关指令是否存在、CPU_NAME、时间戳、开关中断等CPU相关配置
lib_cfg.h 库文件的相关配置
os_cfg.h 系统相关代码配置，该文件是可选的
os_cfg_app.h 系统相关代码配置，该文件是必需的

②应用程序

app.c任务相关代码的编写
app.h任务相关声明、堆栈大小的定义、优先级的设置

③μC/OS-Ⅲ与CPU无关代码

os_cfg_app.c	系统任务的配置
os_type.h	内核对象的数据类型定义
os_dbg.c	调试相关的数据类型定义和相关代码
os_flag.c	事件标志组相关函数
os_int.c	中断延迟相关函数
os_mem.c	内存分区相关函数
os_msg.c	消息相关函数
os_mutex.c	二值信号量相关函数
os_pend_multi.c	等待多个内核对象相关函数
os_prio.c	优先级相关函数
os_q.c	队列相关函数
os_sem.c	多值信号量相关函数
os_stat.c	统计信息相关函数
os_task.c	任务相关函数
os_tick.c	时钟节拍相关函数
os_time.c	时钟管理相关函数
os_tmr.c	定时器相关函数
os_var.c	变量定义相关函数
os.h	相关数据结构类型定义
os_core.c	μC/OS-Ⅲ核心功能相关函数

④库文件

lib_ascii.c
lib_ascii.h
lib_def.h
lib_math.c
lib_math.h
lib_mem_a.asm
lib_mem.c
lib_mem.h
lib_str.c
lib_str.h

⑤μC/OS-Ⅲ与移植相关代码
移植时需要根据不同的CPU进行编写

os_cpu_a.asm	CPU相关的汇编函数定义及声明
os_cpu_c.c	CPU相关的C语言函数定义及声明
os_cpu.h	CPU相关配置及以上两个文件中函数的声明

⑥μC/OS-Ⅲ与CPU相关代码

cpu_def.h	CPU相关配置
cpu_c.c	μC/OS-Ⅲ封装好的CPU相关C语言代码
cpu_a.asm	μC/OS-Ⅲ封装好的CPU相关汇编代码
cpu_core.c	CPU的初始化函数、时间戳计算等CPU核心函数
cpu_core.h	cpu_core.c相关函数的声明

图 10.1 μC/OS-Ⅲ系统文件体系结构

10.2 μC/OS-Ⅲ与 CPU 相关的文件

10.2.1 cpu.c 文件

该文件是与 CPU 底层功能相关的 C 语言文件，主要包括中断控制器和硬件定时器的设置，用户可以选择是否添加。

10.2.2 cpu_a.asm 文件

该文件是与 CPU 底层功能相关的汇编语言文件，该文件主要包括开关中断、数出变量前导零的数目。在任务调度时，使用数出变量前导零的个数操作能够快速确定就绪的最高任务优先级，提高任务调度速度。在该文件中，必须实现保存状态寄存器值的函数 CPU_SR_Save()和恢复状态寄存器值的函数 CPU_SR_Restore()。

10.2.3 cpu_cfg.h 文件

该文件是配置文件，对与 CPU 相关的功能进行裁剪和配置。用户可以直接将该文件复制到工程目录中，根据工程中需要使用的功能对该文件中的宏进行裁剪和配置。cpu_cfg.h 文件代码如下：

```
//定义 CPU 配置模块
#ifndef CPU_CFG_MODULE_PRESENT
#define CPU_CFG_MODULE_PRESENT
//配置 CPU 名称
#define CPU_CFG_NAME_EN DEF_ENABLED
//配置 CPU 名称长度
#define CPU_CFG_NAME_SIZE 16
//不启用 CPU 时间戳配置
#define CPU_CFG_TS_32_EN DEF_DISABLED
#define CPU_CFG_TS_64_EN DEF_DISABLED
//定义 CPU 定时器大小
#define CPU_CFG_TS_TMR_SIZE CPU_WORD_SIZE_32
#define CPU_CFG_INT_DIS_MEAS_OVRHD_NBR 1u
#if 0
//定义汇编语言计算前导零个数
#define CPU_CFG_LEAD_ZEROS_ASM_PRESENT
#endif
#endif
```

10.2.4 cpu_def.h 文件

该文件是宏定义文件，定义了一些和 CPU 配置相关的宏，该文件不需要修改，用户可以直接将其复制到工程目录中使用。cpu_def.h 文件代码如下：

```
//定义 CPU 声明模块
#ifndef CPU_DEF_MODULE_PRESENT
#define CPU_DEF_MODULE_PRESENT
//CPU 内核版本
#define CPU_CORE_VERSION                      13001u
//CPU 字节长度
#define CPU_WORD_SIZE_08                      1u
#define CPU_WORD_SIZE_16                      2u
#define CPU_WORD_SIZE_32                      4u
#define CPU_WORD_SIZE_64                      8u
//CPU 字节序
#define CPU_ENDIAN_TYPE_NONE                  0u
//大端模式
#define CPU_ENDIAN_TYPE_BIG                   1u
```

```
//小端模式
#define CPU_ENDIAN_TYPE_LITTLE                    2u
//堆栈增长方向
#define CPU_STK_GROWTH_NONE                       0u
//堆栈向上生长
#define CPU_STK_GROWTH_LO_TO_HI                   1u
//堆栈向下生长
#define CPU_STK_GROWTH_HI_TO_LO                   2u
#define CPU_CRITICAL_METHOD_NONE                  0u
#define CPU_CRITICAL_METHOD_INT_DIS_EN            1u
#define CPU_CRITICAL_METHOD_STATUS_STK            2u
#define CPU_CRITICAL_METHOD_STATUS_LOCAL          3u
```

10.2.5 cpu.h 文件

该文件定义了在任务编写过程中使用的数据类型。因为每个处理器都存在自身字长，比如 int 类型在 A 处理器上是 4 个字节，在 B 处理器上可能是 2 个字节，所以 μC/OS-Ⅲ操作系统不再使用 char、int、short 等 C 语言传统的基本数据类型，而是使用自己定义的更加明确的基础数据类型。用户可以在移植过程中，根据处理器的类型，在该文件中对数据类型进行定义。cpu.h 文件的主要代码如下所示。

```
//定义各个基本数据类型
typedef                 void            CPU_VOID;
typedef                 char            CPU_CHAR;
typedef     unsigned    char            CPU_BOOLEAN;
typedef     unsigned    char            CPU_INT08U;
typedef       signed    char            CPU_INT08S;
typedef     unsigned    short           CPU_INT16U;
typedef       signed    short           CPU_INT16S;
typedef     unsigned    int             CPU_INT32U;
typedef       signed    int             CPU_INT32S;
typedef     unsigned    long long       CPU_INT64U;
typedef       signed    long long       CPU_INT64S;

typedef                   float         CPU_FP32;
typedef                   double        CPU_FP64;

//定义不需要预编译处理的数据类型
typedef     volatile        CPU_INT08U  CPU_REG08;
typedef     volatile        CPU_INT16U  CPU_REG16;
typedef     volatile        CPU_INT32U  CPU_REG32;
typedef     volatile        CPU_INT64U  CPU_REG64;

typedef                       void                (*CPU_FNCT_VOID)(void);
typedef                       void                (*CPU_FNCT_PTR )(void *p_obj);
```

同时该文件中还定义了一部分宏常量，类似于 cpu_cfg. h 中文件的定义。

```
//配置 CPU 地址字大小
#define CPU_CFG_ADDR_SIZE              CPU_WORD_SIZE_32
//配置 CPU 数据字大小
#define CPU_CFG_DATA_SIZE              CPU_WORD_SIZE_32
//配置 CPU 数据字最大大小
#define CPU_CFG_DATA_SIZE_MAX          CPU_WORD_SIZE_64
//配置 CPU 字节顺序
#define CPU_CFG_ENDIAN_TYPE            CPU_ENDIAN_TYPE_LITTLE
//配置 CPU 堆栈增长类型
#define CPU_CFG_STK_GROWTH             CPU_STK_GROWTH_HI_TO_LO
//配置 CPU 堆栈对齐字节数
#define CPU_CFG_STK_ALIGN_BYTES        (16u)
```

10.2.6 cpu_core. h 文件

该文件主要用于声明 cpu_core. c 文件中定义的函数。

10.2.7 cpu_core. c 文件

该文件是比较上层的 CPU 功能核心库函数文件，主要包括 CPU 初始化函数、用 C 语言实现的前导零个数计算函数、时间戳的设置和获取函数。下面是该文件中实现的几个主要功能函数。

1. void CPU_Init (void)

功能描述：

该函数是 CPU 初始化函数，用来初始化 CPU 时间戳，初始化 CPU 中断关闭时间的测量和 CPU 的主机名称。该函数被应用程序调用来初始化 CPU，需要注意的是，在调用 OS_init()函数之前一定要先调用该函数。

2. void CPU_NameGet (CPU_CHAR *p_name，CPU_ERR *p_err)

参数描述：

(1) pname：指向要存储返回值的 ASCII 码字符数组的指针。

(2) p_err：指向可能的错误代码，具体类型有以下两种。

① CPU_ERR_NONE：函数正常返回，没有出错；

② CPU_ERR_NULL_PTR：p_name 指向了一个空指针。

功能描述：该函数返回 CPU 主机名称。

3. CPU_TS32 CPU_TS_Get32 (void)

功能描述：返回当前 32 位 CPU 的时间戳。

4. CPU_TS_TMR_FREQ CPU_TS_TmrFreqGet (CPU_ERR *p_err)

功能描述：获取时间戳定时器的频率。

5. CPU_DATA CPU_CntLeadZeros（CPU_DATA val）

参数描述：

val：需要计算前导零个数的数据。

功能描述：计算数据前导零的个数。

10.3　μC/OS-Ⅲ系统与 CPU 接口文件

在本章中，假设 CPU 是通用的 ARM32 位处理器。ARM 处理器拥有 16 个通用寄存器（R0～R15），一个中断服务程序堆栈指针寄存器和一个记录 CPU 状态的状态寄存器 SR（State Register）。其中，R0～R3 用作参数寄存器，在函数调用过程中，用于传递参数，当要传递的参数个数大于 4 时，需要使用堆栈来进行参数传递，此时要特别注意堆栈的增长方向。R13 用作链接寄存器（LR）保存函数的返回地址，R14 用作指向任务堆栈的指针，R15 用作 PC 指针，保存处理器将要执行的指令。当任务被中断时，进入中断服务程序，此时堆栈指针由任务堆栈指针自动切换到中断服务程序堆栈指针，保证中断服务程序的正常运行。ARM 寄存器如图 10.2 所示。

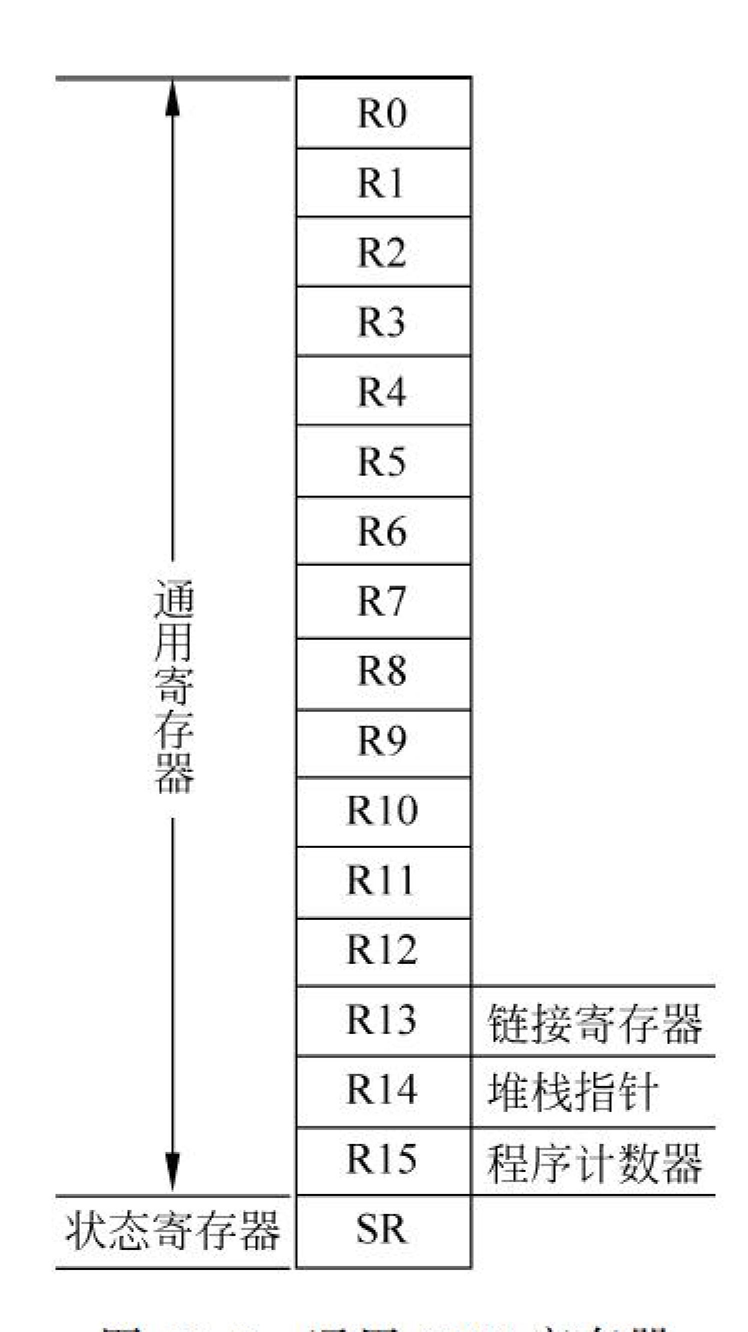


图 10.2　通用 CPU 寄存器

10.3.1　os_cpu.h 文件

该文件声明了一些与 CPU 功能相关的函数，主要包括以下函数。

```
(1) void     OSCtxSw                          (void);    //上下文切换函数;
(2) void     OSIntCtxSw                       (void);    //中断上下文切换函数;
(3) void     OSStartHighRdy                   (void);    //运行最高优先级任务;
(4) CPU_BOOLEAN     OSIntCurTaskSuspend       (void);    //中断处理程序中挂起任务;
(5) CPU_BOOLEAN     OSIntCurTaskResume        (void);    //中断处理程序中唤醒任务;
(6) void OSDebuggerBreak                      (void);    //从调试器中跳出
```

10.3.2 os_cpu_c.c 文件

该文件中主要是一些与 CPU 交互的 C 语言文件,在 μC/OS-Ⅲ官网上下载的源代码文件中包含了模板文件,用户可以根据需要添加函数的实现。下面是一些重要函数的介绍。

1. 函数 OSIdleTaskHook(void)

功能描述:

该函数被系统空闲任务 idle 任务调用,可以在该钩子函数中加入要实现的函数,比如让 CPU 睡眠,从而节省电量。

```
void OSIdleTaskHook(void)
{
#if OS_CFG_APP_HOOKS_EN > 0u
     if (OS_AppIdleTaskHookPtr != (OS_APP_HOOK_VOID)0) {
      (*OS_AppIdleTaskHookPtr)();      //用户定义的钩子函数
     }
#endif

Sleep(1u);                             //减少 CPU 消耗
}
```

同样的函数还有 OSInitHook()、OSStatTaskHook()、OSTaskCreateHook()、OSTaskDelHook()等函数,用户可以在此类函数中添加自己实现的钩子函数,从而完成具体的操作。

2. 函数 CPU_STK *OSTaskStkInit(OS_TASK_PTR p_task

```
void        *p_arg,
CPU_STK     *p_stk_base,
      CPU_STK         *p_stk_limit,
      CPU_STK_SIZE   stk_size,
      OS_OPT         opt)
```

参数描述:

(1) p_task:指向任务的指针;

(2) p_arg:指向任务的输入参数;

(3) p_stk_base:任务堆栈基地址;

(4) p_stk_limit:任务堆栈的限制;

(5) stk_size：任务堆栈大小；

(6) opt：函数选项。

功能描述：

在移植过程中，OSTaskStkInit()函数非常重要，同样也是一个特别容易出错的函数。该函数的主要任务是初始化任务的堆栈。

```
CPU_STK * OSTaskStkInit(OS_TASK_PTR    p_task,
                        void           * p_arg,
                        CPU_STK        * p_stk_base,
                        CPU_STK        * p_stk_limit,
                        CPU_STK_SIZE   stk_size,
                        OS_OPT         opt)
{
   OS_TASK * p_task_info;
   (void)p_stk_limit;    //防止编译器警告
   (void)stk_size;
   //在任务堆栈中创建任务消息结构体
   p_task_info = (OS_TASK * )p_stk_base;
   p_task_info->NextPtr = NULL;
   p_task_info->PrevPtr = NULL;
   p_task_info->OSTCBPtr = NULL;
   p_task_info->OSTaskName = NULL;
   p_task_info->TaskArgPtr = p_arg;
   p_task_info->TaskOpt = opt;
   p_task_info->TaskPtr = p_task;
   p_task_info->TaskState = STATE_NONE;
   p_task_info->ThreadID = 0u;
   p_task_info->ThreadHandle = NULL;
   p_task_info->InitSignalPtr = NULL;
   p_task_info->SignalPtr = NULL;
   return (p_stk_base);
}
```

10.3.3 os_cpu_a.asm 文件

该文件中主要是一些与 CPU 交互的汇编文件，主要包括 OSStartHighRdy()函数、OSCtxSw()函数和 OSIntCtxSw()函数等。

1. OSStartHighRdy()

该函数用于从众多就绪的任务中找出优先级最高的任务执行，该任务被 OSStart()函数调用来管理多个任务的运行。下面给出该函数的示意性代码。

```
OSStartHighRdy:
    OSTaskSwHook();                        //任务切换钩子函数
    SP = OSTCBHighRdyPtr->StkPtr;          //SP 指针指向最高优先级任务的堆栈
    //宏调用,从将要运行的任务堆栈中恢复 CPU 寄存器内容
    OS_CTX_RESTORE
    Return from INterrupt/Exception        //从中断和异常中返回
```

2. OSCtxSw()

该函数用于普通任务的切换,可实现上下文切换和寄存器内容的保存,被 OSTaskSw()函数调用,以下是该函数的示意性代码。

```
OSCtxSw:
    //宏调用,将 CPU 中的寄存器内容保存到任务的堆栈中
    OS_CTX_SAVE
    OSTCBCurPtr->StkPtr = SP                //当前任务的堆栈指针指向 SP 指针
    OSTaskSwHook();                         //任务切换钩子函数
    //将就绪任务的最高优先级赋值给当前任务优先级
    OSPrioCur = OSPrioHighRdy;
    OSTCBCurPtr = OSTCBHighRdyPtr;          //当前任务指针指向最高就绪优先级任务
    SP = OSTCBHighRdyPtr->StkPtr;           //SP 指针指向最高优先级任务的堆栈
    //宏调用,从将要运行的任务堆栈中恢复 CPU 寄存器内容
    OS_CTX_RESTORE
    Return from INterrupt/Exception         //从中断和异常中返回
```

3. OSIntCtxSw()

该函数用于中断任务的切换,被 OSIntExit()函数调用,以下是该函数的示意性代码。

```
OSIntCtxSw:
    OSTaskSwHook();                          //任务切换钩子函数
//将就绪任务的最高优先级赋值给当前任务优先级
    OSPrioCur = OSPrioHighRdy;
    OSTCBCurPtr = OSTCBHighRdyPtr;           //当前任务指针指向最高就绪优先级任务
    SP = OSTCBHighRdyPtr->StkPtr;            //SP 指针指向最高优先级任务的堆栈
    //宏调用,从将要运行的任务堆栈中恢复 CPU 寄存器内容
    OS_CTX_RESTORE
    Return from INterrupt/Exception          //从中断和异常中返回
```

习题

1. 如果要在处理器上移植 μC/OS-Ⅲ操作系统,则该处理器应具备什么条件?
2. μC/OS-Ⅲ移植需要修改哪些文件?

3. cpu_cfg.h 文件主要配置了什么变量？cpu_def.h 文件呢？

4. cpu_core.c 文件是移植过程中与 CPU 确切相关的核心文件，请详细说明该文件的功能及其中的函数功能。

5. 通用 ARM 处理器各个寄存器的功能是什么？

6. 任务栈初始化是如何进行的？

参 考 文 献

［1］ Jean J. Labrosse.嵌入式实时操作系统 μC/OS-Ⅲ［M］.宫辉，曾明，龚光华，等译.北京：北京航空航天大学出版社，2012.

［2］ Jean J. Labrosse.嵌入式实时操作系统 μC/OS-Ⅲ应用开发.何小庆，张爱华，译.北京：北京航空航天大学出版社，2012.

［3］ 李悦城，野火，刘焿.μC/OS-Ⅲ源码分析笔记［M］.北京：机械工业出版社，2016.

［4］ 卢有亮.嵌入式实时操作系统 μC/OS 原理与实践［M］.2 版.北京：电子工业出版社，2014.

［5］ 任哲，房红征，曹靖.嵌入式实时操作系统 μC/OS-Ⅱ原理及应用［M］.4 版.北京：北京航空航天大学出版社，2016.

［6］ 刘波文，孙岩.嵌入式实时操作系统 μC/OS-Ⅱ经典实例：基于 STM32 处理器［M］.2 版.北京：北京航空航天大学出版社，2014.